

Ein kleiner Überblick über

Neuronale Netze

David Kriesel

dkriesel.com

Downloadadresse:
http://www.dkriesel.com/science/neural_networks

NEU - Für die Coder:
Skalierbares, effizientes NN-Framework für JAVA
http://www.dkriesel.com/tech/snipe

In Gedenken an
Dr. Peter Kemp, Notar a.D., Bonn.

Vorwörtchen

„Diese Arbeit ist ursprünglich im Rahmen eines Seminars der Rheinischen Friedrich-Wilhelms-Universität Bonn entstanden, wurde und wird jedoch (nachdem vorgetragen und online verfügbar gemacht unter `www.dkriesel.com` am 27.05.2005) immer mehr erweitert – erstens, um mit der Zeit einen möglichst guten Überblick über das Thema der Neuronalen Netze liefern zu können und zweitens schlicht und einfach, weil ich daran immer mehr und besser LaTeX lernen möchte. Wer weiss, vielleicht wird aus dieser Zusammenfassung ja sogar einmal ein Vorwort!“

Zusammenfassung dieser Arbeit, Ende 2005

Aus der obenstehenden Zusammenfassung ist bis jetzt zwar kein Vorwort, aber immerhin ein *Vorwörtchen* geworden, nachdem sich die Erweiterung der Arbeit (damals auf 40 Seiten) bei mir unverhofft zum Downloadschlager entwickelt hat.

Anspruch und Intention dieses Manuskripts

Der ganze Text ist großzügiger geschrieben und ausführlicher bebildert als früher. Die Illustrationen sind nun „nach meinem Gusto“ selbst erstellt, zum Großteil direkt in LaTeX unter Verwendung des Pakets XYpic. Sie spiegeln das wieder, was ich mir beim Erarbeiten des Stoffs gewünscht hätte, sollen also mit dem Text zusammen möglichst eingängig und schnell verständlich sein, um möglichst vielen einen Einstieg in das Gebiet der Neuronalen Netze zu ermöglichen.

Trotzdem kann der mathematisch und formal versierte Leser die Definitionen weitestgehend ohne Fließtext lesen, der nur an dem Gebiet an sich interessierte Leser umgekehrt; alles wird also sowohl umgangssprachlich wie auch formal erklärt. Ich bitte ausdrücklich um Hinweise, falls ich diese Doppelmoral einmal nicht ausreichend bewerkstelligt haben sollte.

Abschnitte dieser Arbeit sind weitestgehend eigenständig

Das Dokument selbst gliedert sich in mehrere Teile, die sich wiederum in Kapitel teilen. Trotz Querverweisen untereinander kann man die Kapitel mit nur geringem Vorwissen auch für sich selbst lesen. Hierbei gibt es größere und kleinere Kapitel: Während die größeren einen etwas fundierteren Einblick in ein Paradigma Neuronaler Netze geben sollen (z.B. beim Standardbeispiel Neuronaler Netze, dem *Perceptron* und seiner Lernverfahren), sind die kleineren für einen kurzen Überblick gedacht – dies wird in der jeweiligen Kapiteleinleitung auch beschrieben. In einigen Exkursen möchte ich noch gerne im Zusammenhang interessantes, aber nicht direkt dem Stoff zugehöriges Wissen vermitteln.

Auf der Suche nach kostenlosen deutschen Quellen, die inhaltlich (was die Paradigmen Neuronaler Netze angeht) vielfältig, aber trotzdem durchgehend einheitlichen Stils sind, bin ich (zumindest was deutschsprachiges Material angeht) leider nicht wirklich fündig geworden. Das Ziel dieser Arbeit (auch wenn sie es vielleicht nicht auf Anhieb erfüllen kann) ist, diese Lücke nach und nach zu schließen und das Fachgebiet auf leicht verständliche Weise zugänglich zu machen.

Für diejenigen, die direkt während des Lesens programmieren möchten, gibt es SNIPE

SNIPE[1] ist eine ausführlich dokumentierte JAVA-Bibliothek, welche ein schnelles, feature-reiches und einfach benutzbares Framework für Neuronale Netze implementiert. Für nichtkommerzielle Einsatzgebiete ist es kostenlos verfügbar. Es war von mir ursprünglich für den Einsatz in Hochleistungssimulationen konzipiert, in denen sehr viele, auch große Netze gleichzeitig trainiert und ausgeführt werden. Vor kurzem habe ich mich nun entschieden, Snipe als professionelle Referenzimplementierung zu dieser Arbeit online zu stellen, die sehr viele der behandelten Aspekte abdeckt, aber angesichts des ursprünglichen Design-Ziels effizienter arbeitet als die meisten anderen Implementierungen. Wer also direkt vieles vom Gelernten ausprobieren möchte oder aus anderen Gründen nach dem Lernen eine schnelle und stabile Neuronale-Netze-Implementierung braucht, ist mit Snipe sicher gut beraten.

Die von Snipe abgedeckten Aspekte sind allerdings nicht komplett deckungsgleich mit dem Manuskript. Manche im Manuskript vorgestellten Arten Neuronaler Netze werden von Snipe nicht unterstützt, während Snipe in anderen Bereichen deutlich mehr

1 Scalable and Generalized Neural Information Processing Engine, Download unter `http://www.dkriesel.com/tech/snipe`, Online-JavaDoc unter `http://snipe.dkriesel.com`

Fähigkeiten hat, als im Manuskript in Form von Praxistipps beschrieben werden kann (ich habe aber die Erfahrung gemacht, dass die allermeisten Implementierungsinteressen der Leser gut abgedeckt werden). Auf der Snipe-Downloadseite gibt es in der Rubrik „Loslegen mit Snipe“ eine einfache, schrittweise Einführung in Snipe und seine Dokumentation, sowie einige Beispiele.

SNIPE: Dieses Manuskript integriert Snipe sehr stark. Über weite Teile des Manuskriptes hinweg finden sich abgesetzte, unterlegte Textstellen wie diese, aus denen hervorgeht, wie der gerade beschriebene Kontext in Snipe realisiert werden kann. **Wer Snipe nicht nutzen möchte, muss also nur diese abgesetzten Textstellen überspringen!** Die Snipe-Textstellen setzen voraus, dass man sich vorher die „Loslegen mit Snipe“-Anleitung auf der Downloadseite zu Gemüte geführt hat. Oftmals werden Klassennamen verwendet; da Snipe nicht sehr viele Java-Pakete hat, habe ich die Paketnamen der Übersichtlichkeit halber nicht den Klassennamen vorangestellt.

Es ist einfach, diese Arbeit zu drucken

Diese Ausarbeitung ist durchgehend farbig gehalten, kann jedoch auch so, wie sie ist monochrom gedruckt werden: Die Farben sind in Abbildungen, Tabellen und Text so gewählt, dass sie neben schönem Farbdesign auch einen hervorragenden Schwarz-Weiss-Kontrast ergeben.

Es sind viele Hilfsmittel im Text integriert

In das Dokument sind verschiedene Hilfsmittel direkt integriert, die das Lesen wesentlich flexibler machen sollen: Wer (wie ich) lieber auf Papier liest als am Bildschirm, soll sich trotzdem einiger Features erfreuen.

Verschiedene Kapiteltypen sind im Inhalt gekennzeichnet

Direkt im Inhaltsverzeichnis sind die Kapiteltypen gekennzeichnet. Kapitel, die als „wichtige Grundlagen“ gekennzeichnet sind, sollte man definitiv zu Anfang lesen, denn eigentlich alle nachfolgenden Kapitel basieren in irgendeiner Form darauf. Andere Kapitel basieren zusätzlich auf bestimmten anderen (vorherigen) Kapiteln, auch hier steht eine entsprechende Bemerkung im Inhaltsverzeichnis neben dem Kapiteltitel.

Sprechende Überschriften im Text, kürzere im Inhaltsverzeichnis

Das ganze Scriptum ist nun mit sprechenden Überschriften durchzogen. Sprechende Überschriften sind nicht nur einfach ein Titel wie z.B. „Bestärkendes Lernen“, sondern definieren den Kernpunkt des zugehörigen Abschnitts: in diesem Fall „Bestärkendes Lernen gibt dem Netz Feedback, ob es sich gut oder schlecht verhält“. Die letztere, lange Version dient hierbei als Überschrift, die im Text verwendet wird, die kürzere steht im Inhaltsverzeichnis, so dass dieses griffig bleibt.

Randbemerkungen sind eine Navigationshilfe

Über das ganze Dokument hinweg existieren umgangssprachliche Randhinweise (siehe nebenstehendes Beispiel) , an denen entlang man (unter Mitbenutzung der Überschriften) durch den Text „gleiten“ und Textstellen einfach wiederfinden kann.

Neue mathematische Symbole kennzeichne ich zum einfachen Wiederfinden mit besonderen Randhinweisen (nebenstehend ein Beispiel für x).

Es gibt verschiedene Arten der Indizierung

Es existieren verschiedene Arten der Indizierung: Zunächst kann man einen Begriff, nachdem man ihn im Index gefunden und die betreffende Seite aufgeschlagen hat, einfach finden, indem man nach ***hervorgehobenem Text*** sucht – indizierte Begriffe sind grundsätzlich auf diese Weise hervorgehoben.

Kapitelübergreifende mathematische Symbole (wie z.B. Ω für ein Outputneuron, ich habe mich bemüht, bei allgegenwärtig wiederkehrenden Elementen eine konsistente Nomenklatur beizubehalten) besitzen eine eigene Indexkategorie unter „Mathematische Symbole“, so dass sie einfach dem entsprechenden Begriff zugeordnet werden können.

Personennamen, welche in Kapitälchen geschrieben sind, werden in der Indexkategorie „Personen“ indiziert und nach Nachnamen geordnet.

Nutzungsbedingungen und Lizenz

Von der Epsilon-Edition an ist das Manuskript unter *Creative Commons Attribution-No Derivative Works 3.0 Unported License*[2] lizensiert, bis auf einige wenige Kleinteile,

2 http://creativecommons.org/licenses/by-nd/3.0/

die liberaleren Lizenzen unterstehen (im Wesentlichen ein paar Bilder, die in den Wikimedia Commons sind). Hier ist eine Kurzzusammenfassung dessen, was diese Lizenz ungefähr wiedergibt:

1. Dieses Dokument darf frei weiterverbreitet werden (auch wenn es eine bessere Idee ist, einfach die URL meiner Homepage weiterzuverbreiten, denn hier gibt es schließlich immer die neueste Version).
2. Dieses Dokument darf nicht modifiziert oder als Teil eines anderen verwendet werden, insbesondere nicht für kommerzielle Zwecke.
3. Dieses Dokument muss in jeglicher seiner Verwendungen dem Autor zugeschrieben werden. Die Urheberschaft des Autors darf nicht verschleiert werden.
4. Die o.g. Zuschreibung des Dokumentes zum Autor impliziert nicht, dass der Autor die Art befürwortet, auf die ein beliebiger Leser das Dokument nutzt.

Da ich kein Anwalt bin, ist die obige Stichpunktzusammenfassung nur informativ gemeint. Wenn sie in irgendeiner Form in Konflikt zur o.g. Creative Commons-Lizenz steht, so hat letztere in jedem Fall Vorrang. Natürlich berührt die Lizenz ebenfalls nicht den Source Code des Manuskripts, der nicht veröffentlicht wird.

Wie dieser Text zitiert wird

Da dieser Text keinen offiziellen Verlag hat, muss man mit Referenzen sorgfältig sein: Hierzu gibt es Informationen in Deutsch und Englisch auf meiner Homepage bzw. der zum Text gehörenden Unterseite[3].

Danksagung

Ich möchte nun einige Danksagungen loswerden, da ein Skriptum wie dieses durchaus genug Arbeit macht, um viele Helfer zu benötigen. Zunächst einmal möchte ich mich bei den Korrektoren dieses Skriptums bedanken, welche mir und der Leserschaft sehr sehr geholfen haben. Genannt seien in alphabetischer Ordnung: Wolfgang Apolinarski, Kathrin Gräve, Paul Imhoff, Thomas Kühn, Christoph Kunze, Malte Lohmeyer, Joachim Nock, Daniel Plohmann, Daniel Rosenthal, Christian Schulz und Tobias Wilken.

Vielen Dank für Verbesserungen, Feedback und Anmerkungen möchte ich auch den Lesern Dietmar Berger, Igor Buchmüller, Marie Christ, Julia Damaschek, Jochen Döll,

3 http://www.dkriesel.com/science/neural_networks

Maximilian Ernestus, Hardy Falk, Anne Feldmeier, Sascha Fink, Andreas Friedmann, Jan Gassen, Markus Gerhards, Sebastian Hirsch, Andreas Hochrath, Nico Höft, Thomas Ihme, Boris Jentsch, Tim Hussein, Thilo Keller, Mario Krenn, Mirko Kunze, Maikel Linke, Adam Maciak, Benjamin Meier, David Möller, Andreas Müller, Rainer Penninger, Lena Reichel, Alexander Schier, Matthias Siegmund, Mathias Tirtasana, Oliver Tischler, Maximilian Voit, Igor Wall, Achim Weber, Frank Weinreis, Gideon Maillette de Buij Wenniger, Philipp Woock und vielen anderen aussprechen.

Herzlicher Dank geht an Sebastian Merzbach, der die Epsilon2-Version des Skriptums auf gewissenhafteste und gründlichste Art und Weise auf Inkonsistenzen und Fehler durchgesehen hat. Insbesondere die englische Version verdankt ihm zusätzlich unzählige sprachliche und ortographische Verbesserungen.

Ebenfalls danke ich Beate Kuhl ganz besonders für die Übersetzung des Skriptums vom Deutschen ins Englische und Nachfragen, die mich bewogen haben, einige Formulierungen anders zu gestalten.

Ganz besonderen Dank möchte ich Prof. Rolf Eckmiller und Dr. Nils Goerke aussprechen, sowie der ganzen Abteilung Neuroinformatik des Instituts für Informatik der Universität Bonn – sie sorgten dafür, dass ich immer Neues über Neuronale Netze und Fachverwandtes lernte (und auch lernen musste). Insbesondere Herr Dr. Goerke war und ist immer bereit, auf jedwede Frage einzugehen, die ich mir während des Schreibens nicht selber beantworten konnte. Gespräche mit Prof. Eckmiller waren für mich immer bereichernd und gaben Anstoß für weitere Recherchen oder verleiteten mich, „von der Tafel zurückzutreten“, um Sachverhalte von weiter weg nüchtern zu betrachten und nächste Aufgaben auszumachen.

Ganz global und nicht nur auf das Skriptum bezogen geht ein weiteres Dankeschön an meine Eltern, welche nie müde werden, mir fachbezogene und damit nicht wirklich preiswerte Buchwünsche zu erfüllen und mir auch sonst viel Unterstützung für mein Studium zukommen lassen.

Für viele „Bemerkungen“ und die ganz besonders herzliche Stimmung ;-) bedanke ich mich sehr bei Andreas Huber und Tobias Treutler, mit denen es seit dem ersten Semester nur selten langweilig wurde!

Ich möchte auch noch kurz an meine Schulzeit denken und mich bei jenen Lehrern ganz herzlich danken, die mir (das ist meine Überzeugung) trotz meiner vielleicht nicht immer vollherzigen Mitarbeit naturwissenschaftlich etwas mit auf den Weg gegeben haben: Herrn Wilfried Hartmann, Herrn Hubert Peters und Herrn Frank Nökel.

Ein Dankeschön geht auch an die Wikimedia Commons, wo ich einige (wenige) Bildvorlagen entnommen und auf mein Skriptum angepasst habe.

Weiterhin danke ich der gesamten Mann- bzw. Frauschaft des Notariates Dr. Kemp Dr. Kolb aus Bonn, bei der ich mich immer gut aufgehoben fühle und die alles tut, damit sich meine Druckkosten im Rahmen halten - insbesondere seien hier erwähnt Christiane Flamme und Herr Dr. Kemp!

Als letztes, und sozusagen als Ehrenplatz, möchte ich aber zwei Personen danken, die sich sehr um das Skriptum verdient machen: Meine Freundin Verena Thomas, die, obwohl viele andere Dinge für sie zu tun sind, viele mathematische und logische Fehler in meinem Skriptum aufgedeckt und mit mir diskutiert hat – und Christiane Schultze, die das Skript sorgfältigst auf Rechtschreibfehler und Inkonsistenzen durchgesehen hat.

David

David Kriesel

Inhaltsverzeichnis

Teil I

Von der Biologie zur Formalisierung – Motivation, Philosophie, Geschichte und Realisierung Neuronaler Modelle

Kapitel 1

Einleitung, Motivation und Geschichte

Wie kann man dem Computer etwas beibringen? Entweder, indem man ein starres Programm schreibt – oder, indem man ihm das Lernen ermöglicht. Lebende Wesen haben keinen Programmierer, der ihnen ein Programm für ihre Fähigkeiten schreibt, welches nur ausgeführt werden muss. Sie lernen – ohne Vorkenntnisse durch Eindrücke von außen – selber und kommen damit zu besseren Problemlösungen als jeder heutige Computer. Kann man solches Verhalten auch bei Maschinen wie Computern erreichen? Können wir solche Kognition von der Biologie adaptieren? Geschichte, Entwicklung, Niedergang und Wiederauferstehung eines großen Ansatzes, Probleme zu lösen.

1.1 Warum Neuronale Netze?

Es gibt Kategorien von Problemen, welche sich nicht in einen Algorithmus fassen lassen – Probleme, die von einer großen Menge subtiler Faktoren abhängen, wie zum Beispiel die Bestimmung des Kaufpreises einer Immobilie, den wir mit unserem Gehirn (ungefähr) bestimmen können, ein Computer in Ermangelung eines Algorithmus aber nicht. Darum muss man sich die Frage stellen: *Wie lernen wir denn, auf solche Fragestellungen einzugehen?*

Genau – wir *lernen*; eine Fähigkeit, an der es Computern offensichtlich mangelt . Menschen haben ein Gehirn mit der Fähigkeit zu lernen, Computer einige Recheneinheiten und Speicher. Diese ermöglichen, in kürzester Zeit komplizierteste numerische Berechnungen auszuführen, bieten uns jedoch keine Lernfähigkeit. Stellen wir Computer und

	Gehirn	**Computer**
Anzahl Recheneinheiten	$\approx 10^{11}$	$\approx 10^{9}$
Art Recheneinheiten	Neurone	Transistoren
Art der Berechnung	massiv parallel	i.d.R. seriell
Datenspeicherung	assoziativ	adressbasiert
Schaltzeit	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Theoretische Schaltvorgänge	$\approx 10^{13}\frac{1}{s}$	$\approx 10^{18}\frac{1}{s}$
Tatsächliche Schaltvorgänge	$\approx 10^{12}\frac{1}{s}$	$\approx 10^{10}\frac{1}{s}$

Tabelle 1.1: Der (hinkende) Vergleich zwischen Gehirn und Rechner auf einen Blick. **Vorlage:** [Zel94]

Gehirn gegenüber[1], so bemerken wir, dass der Computer theoretisch leistungsfähiger sein müsste: Er besitzt 10^9 Transistoren mit einer Schaltzeit von 10^{-9} Sekunden. Das Gehirn hat zwar 10^{11} Neurone, jedoch schalten diese nur in etwa 10^{-3} Sekunden.

Allerdings arbeitet der größte Teil des Gehirns durchgehend, während wiederum der größte Teil des Computers nur passiv Daten speichert. So arbeitet das Gehirn parallel und damit nahe an seiner theoretischen Maximalleistung, der Computer ist von dieser jedoch um Zehnerpotenzen entfernt (Tabelle 1.1). Zudem ist ein Computer in sich statisch - das Gehirn als biologisches Neuronales Netz kann sich jedoch während seiner „Laufzeit“ umstrukturieren und so lernen, Fehler kompensieren und mehr.

Innerhalb dieser Arbeit möchte ich skizzieren, wie man sich solche Eigenschaften des Gehirns auch am Computer zunutze macht.

Das Studium der Künstlichen Neuronalen Netze ist also motiviert durch die Ähnlichkeit zu erfolgreich arbeitenden biologischen Systemen, welche im Vergleich zum Gesamtsystem aus sehr einfachen , aber dafür vielen und massiv parallel arbeitenden Nervenzellen bestehen und (das ist wohl einer der bedeutendsten Aspekte) ***Lernfähigkeit*** besitzen. Ein Neuronales Netz muss nicht explizit für seine Aufgaben programmiert werden, es kann beispielsweise aus Trainingsbeispielen lernen oder auch durch Bestärkung, sozusagen durch Zuckerbrot und Peitsche (*Reinforcement Learning*).

Ein aus dem Lernvorgang resultierender Aspekt ist die ***Generalisierungs- bzw. Assoziationsfähigkeit*** Neuronaler Netze: Nach erfolgreichem Training kann ein Neu-

1 Diese Gegenüberstellung ist natürlich aus vielen naheliegenden Gründen bei Biologen wie Informatikern umstritten, da Schaltzeit und Menge nichts über Qualität und Leistung der Recheneinheiten aussagt und Neurone und Transistoren auch nicht direkt vergleichbar sind. Dennoch erfüllt die Gegenüberstellung den Zweck, den Vorteil der Parallelität anhand der Verarbeitungszeit aufzuzeigen.

ronales Netz ähnliche Probleme derselben Klasse, die nicht explizit trainiert wurden, plausiblen Lösungen zuführen. Daraus resultiert dann wieder eine große ***Fehlertoleranz*** gegenüber verrauschten Eingabedaten.

Fehlertoleranz steht wieder in enger Beziehung zu biologischen Neuronalen Netzen, bei denen diese Eigenschaft sehr ausgeprägt ist: Wie schon bemerkt, hat ein Mensch ca. 10^{11} Neurone, die sich aber kontinuierlich umstrukturieren oder durch Einflüsse von außen umstrukturiert werden (bei einem Vollrausch verliert ein Mensch ca. 10^5 Neurone, auch bestimmte Arten Nahrungsmittel oder Umwelteinflüsse zerstören Gehirnzellen) – trotzdem wird unsere Kognitionsfähigkeit nicht wesentlich beeinträchtigt. Das Gehirn ist also tolerant gegenüber inneren Fehlern – und auch gegenüber Fehlern von außen, denn so manche „Sauklaue“ können wir immer noch lesen, obwohl einzelne Buchstaben vielleicht gar nicht wirklich auszumachen sind.

Unsere moderne Technologie hingegen ist noch nicht automatisch fehlertolerant – mir ist noch kein Computer bekannt, in dem jemand vergessen hat, den Festplattencontroller einzubauen, weshalb die Grafikkarte automatisch dessen Job übernimmt, Leiterbahnen ausbaut und Kommunikation entwickelt, so dass der fehlende Baustein das Gesamtsystem nur wenig beeinträchtigt, aber auf keinen Fall völlig lahmlegt.

Nachteil dieser verteilten, fehlertoleranten Speicherung ist natürlich, dass wir einem Neuronalen Netz nicht ohne weiteres ansehen können, was es weiß, kann oder wo seine Fehler liegen – Analysen dieser Art sind bei herkömmlichen Algorithmen in der Regel wesentlich leichter. Auch bekommen wir das Wissen in unser Neuronales Netz meistens nur durch einen *Lernvorgang*, bei dem verschiedene Fehler passieren können und der nicht immer einfach zu handhaben ist.

Fehlertoleranz von Daten ist bei der aktuellen Technologie schon wesentlich ausgereifter: Vergleichen wir eine Schallplatte mit einer CD. Ist auf der Schallplatte ein Kratzer, so ist die Toninformation an der Stelle des Kratzers für einen winzigen Moment komplett verloren (man hört ein Knacken), danach geht die Musik weiter. Bei einer CD sind die Audiodaten *verteilt* gespeichert: Ein Kratzer sorgt für einen unschärferen Ton in seiner näheren Umgebung, der Datenstrom an sich bleibt aber weitestgehend unbeeinträchtigt – mit der Folge, dass der Hörer ihn nicht bemerkt.

Wir halten also die herausragenden Merkmale fest, die wir aus der Biologie zu adaptieren versuchen:

- ▷ Selbstorganisation bzw. Lernfähigkeit,
- ▷ Generalisierungsfähigkeit und
- ▷ Fehlertoleranz.

Welche Arten von Neuronalen Netzen welche Fähigkeiten besonders stark ausbilden, welche für was für Problemklassen nutzbar sind, werden wir im Verlauf dieser Arbeit noch herausfinden.

Direkt im Einleitungskapitel dieser Arbeit sei gesagt: „*Das* Neuronale Netz“ gibt es so nicht. Es gibt viele verschiedene Paradigmen, was Neuronale Netze sind, wie sie trainiert und wo sie eingesetzt werden – und mein Ziel ist es, einige dieser Paradigmen anschaulich vorzustellen und mit Bemerkungen für die praktische Anwendung zu versehen.

Oben haben wir bereits erwähnt, dass die Arbeit z.B. unseres Gehirns im Gegensatz zur Arbeit eines Computers massiv parallel stattfindet, also jeder Baustein zu jeder Zeit aktiv ist. Möchte man nun ein Argument für den Nutzen massiver Parallelverarbeitung anführen, so wird oft die ***100-Schritt-Regel*** genannt.

1.1.1 Die 100-Schritt-Regel

Durch Experimente hat man festgestellt, dass ein Mensch ein Bild eines bekannten Gegenstandes bzw. einer bekannten Person in ≈ 0.1 Sekunden erkennen kann, also bei einer Neuronenschaltzeit von $\approx 10^{-3}$ Sekunden in ≈ 100 diskreten Zeitschritten ***paralleler*** Verarbeitung.

Ein der Von-Neumann-Architektur folgender Rechner hingegen kann in 100 Zeitschritten *sequentieller* Verarbeitung, also beispielsweise 100 Assemblerschritten oder Taktschritten, so gut wie nichts tun.

Nun wollen wir ein einfaches Anwendungsbeispiel betrachten, bei dem ein Neuronales Netz zum Einsatz kommen könnte.

1.1.2 Einfache Anwendungsbeispiele

Angenommen, wir besitzen einen kleinen Roboter wie in Abb. 1.1 auf der rechten Seite. Dieser Roboter besitzt acht Abstandssensoren, aus denen er Eingabedaten gewinnt: Drei Sensoren vorne rechts, weitere drei vorne links, und zwei hinten. Jeder dieser Sensoren liefert uns zu jeder Zeit einen reellen Zahlenwert, wir erhalten also immer einen Input $I \in \mathbb{R}^8$.

In unserem einfachen Beispiel kann der Roboter trotz seiner zwei Motoren (die brauchen wir später) noch nicht viel: Er soll einfach immer fahren, aber anhalten, wenn er Gefahr läuft an ein Hindernis zu stoßen. Unser Output ist also binär: $H = 0$ für

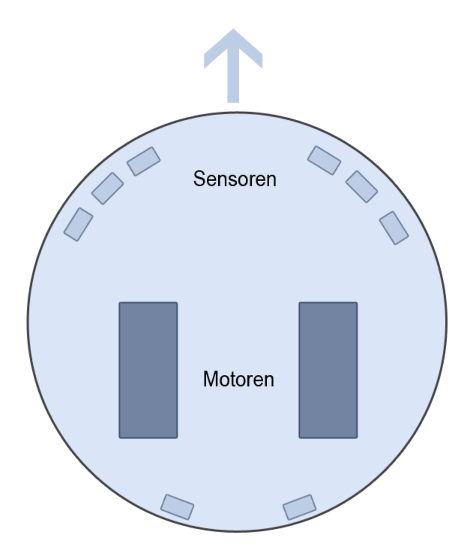


Abbildung 1.1: Ein kleiner Roboter mit acht Sensoren und zwei Motoren. Der Pfeil zeigt die Fahrtrichtung an.

„Alles okay, fahr weiter“ und $H = 1$ für „Halte an“ (Wir nennen den Output H für „Haltesignal“). Wir benötigen also eine Abbildung

$$f : \mathbb{R}^8 \to \mathbb{B}^1,$$

welche die Inputsignale einer Robotertätigkeit zuführt.

1.1.2.1 Der klassische Weg

Es gibt nun zwei Wege, diese Abbildung zu realisieren. Zum einen gibt es den *klassischen Weg*: Wir setzen uns eine Weile hin und denken nach, so dass wir am Ende eine Schaltung oder ein kleines Computerprogramm erhalten, dass die Abbildung realisiert (bei der Einfachheit des Beispieles ist das natürlich ohne weiteres möglich). Anschließend nehmen wir die technischen Referenzen der Sensoren zur Hand, studieren die Kennlinie der Sensoren, um zu wissen, was für Werte bei welchen Hindernisentfernungen ausgegeben werden, und binden die Werte in unser oben gebautes Regelwerk ein. Solche Verfahren finden in der klassischen Künstlichen Intelligenz Anwendung, und falls man die Regeln für eine Abbildung exakt kennt, ist man auch gut beraten, diesem Schema zu folgen.

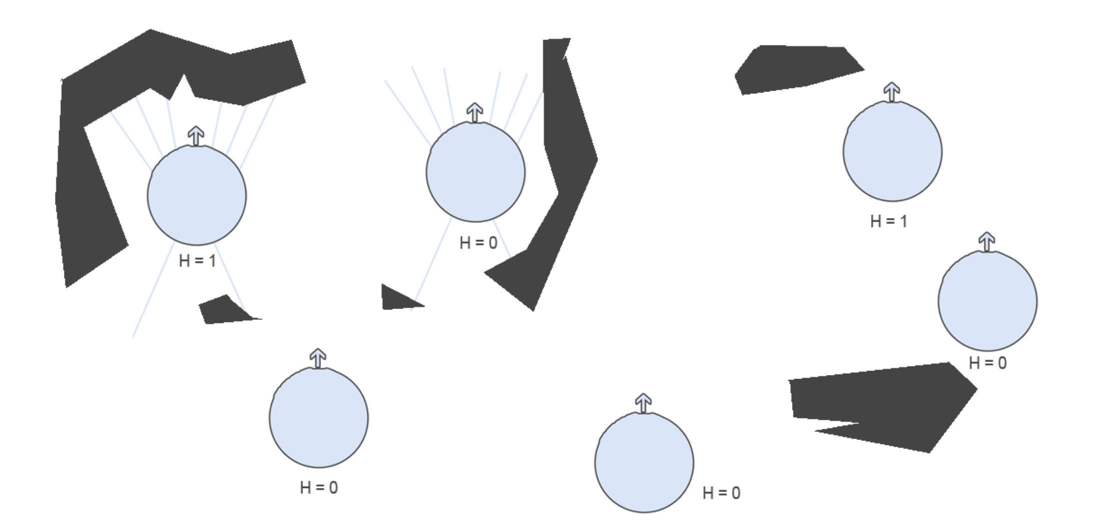


Abbildung 1.2: Der Roboter wird in eine Landschaft platziert, die ihm Sensorwerte für verschiedene Situationen liefert. Die gewünschten Ausgabewerte H geben wir selbst hinzu, und erhalten so unsere Lernbeispiele. Bei zwei Robotern sind exemplarisch die Richtungen aufgetragen, in die die Sensoren weisen.

1.1.2.2 Der Weg des Lernens

Für uns hier interessanter und für viele Abbildungen und Probleme, welche nicht auf Anhieb erfassbar sind, auch erfolgreicher ist aber der *Weg des Lernens*: Wir zeigen dem Roboter verschiedene Situationen (Abb. 1.2), in denen er sich beispielsweise befinden kann – und er soll selbst lernen, was in seinem Roboterleben zu tun ist.

In diesem Beispiel soll er einfach lernen, wann anzuhalten ist. Wir stellen uns hierzu erst einmal das Neuronale Netz als eine Art ***Black Box*** (Abb. 1.3 auf der rechten Seite) vor, kennen also nicht seinen Aufbau, sondern betrachten es rein in seinem Verhalten nach außen.

Die Situationen in Form von einfach gemessenen Sensorwerten (wir stellen den Roboter z.B. einfach vor ein Hindernis, siehe Abbildung), die wir dem Roboter zeigen und für die wir ihm vorgeben, ob weiterzufahren oder zu halten ist, nennen wir *Trainingsbeispiele* – ein Trainingsbeispiel besteht also aus einem beispielhaften Input und einem dazugehörigen gewünschten Output. Die Frage ist nun, wie wir dieses Wissen, die Information, in das Neuronale Netz transportieren.

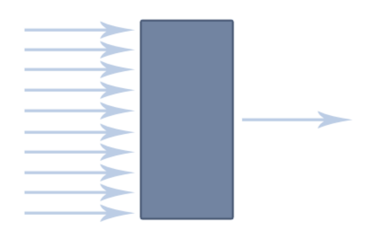


Abbildung 1.3: Wir betrachten die Robotersteuerung zunächst als Black Box, deren Innenleben uns unbekannt ist. Die Black Box nimmt einfach acht reelle Sensorwerte entgegen und bildet diese auf einen binären Ausgabewert ab.

Die Beispiele lassen sich durch ein einfaches *Lernverfahren* einem Neuronalen Netz beibringen (ein Lernverfahren ist ein einfacher Algorithmus bzw. eine mathematische Formel). Hier wird das Neuronale Netz, wenn wir alles richtig gemacht und gute Beispiele gewählt haben, aus den Beispielen *generalisieren* und so eine allgemeingültige Vorschrift finden, wann anzuhalten ist.

Unser Beispiel ist beliebig erweiterbar – die Motoren unseres Roboters könnten auch zwecks Richtungssteuerung separat steuerbar sein[2], bei ansonsten gleichem Sensorlayout. In diesem Fall suchen wir eine Abbildung

$$f : \mathbb{R}^8 \to \mathbb{R}^2,$$

welche die beiden Motoren anhand der Sensorinputs stufenlos steuert und so den Roboter z.B. nicht nur anhalten, sondern auch Hindernissen ausweichen lassen kann – hier ist es schon schwieriger, aus dem Kopf Regeln abzuleiten, und de facto ein Neuronales Netz angebrachter.

Ziel ist es also nicht etwa, die Beispiele auswendig zu lernen – sondern das *Prinzip* dahinter zu realisieren: Der Roboter soll das Neuronale Netz im Idealfall in beliebigen Situationen anwenden und Hindernissen ausweichen können. Insbesondere soll der Roboter das Netz *während* des Fahrens kontinuierlich bzw. oft direkt hintereinander befragen können, um kontinuierlich Hindernissen auszuweichen. Dies ergibt einen ständigen Kreislauf: Der Roboter befragt das Netz. Er fährt dadurch in eine Richtung, wodurch sich seine Sensorwerte verändern. Er befragt wieder das Netz und verändert abermals seine Position, die Sensorwerte verändern sich erneut, und so weiter. Dass

2 Es gibt einen Roboter namens *Khepera*, der in etwa diese Eigenschaften besitzt. Er ist rund, hat ca. 7cm Durchmesser, besitzt zwei Motoren mit Rädern und verschiedene Sensoren. Zur Veranschaulichung kann ich nur empfehlen, einmal im Internet danach zu recherchieren.



Abbildung 1.4: Einige Urgesteine des Fachbereichs der Neuronalen Netze. Von links nach rechts: John von Neumann, Donald O. Hebb, Marvin Minsky, Bernard Widrow, Seymour Papert, Teuvo Kohonen, John Hopfield, alle weitestgehend „in order of appearance“.

sich dieses System auch auf dynamische, das heißt sich selbst verändernde Umwelten (z.B. bewegte Hindernisse in unserem Beispiel) adaptieren lässt, ist ersichtlich.

1.2 Zur Geschichte Neuronaler Netze

Wie jedes Naturwissenschaftliche Gebiet hat auch die Lehre der Neuronalen Netze eine ***Entwicklungsgeschichte*** hinter sich, welche, wie wir gleich sehen werden, Höhen und Tiefen aufweist. Dem Stil meiner Arbeit treu bleibend stelle ich diese Geschichte nicht als Text, sondern kompakter als Zeitleiste dar. Zitat- und Literaturangaben werde ich hier im Wesentlichen bei Punkten niederschreiben, welche im weiteren Skriptum nicht mehr bearbeitet werden. Zu Stichworten, die wir später noch genauer ergründen, werden die Zitatangaben in den entsprechenden Kapiteln geliefert.

Die Geschichte nimmt ihren Anfang in den frühen 1940er Jahren und damit fast zeitgleich mit der Geschichte der programmierbaren elektronischen Computer. Wie auch bei der Informatik an sich erkennt man die Jugend des Gebiets daran, dass viele der erwähnten Personen auch heute noch unter uns weilen.

1.2.1 Anfänge

Bereits 1943 beschreiben Warren McCulloch und Walter Pitts eine Art neurologischer Netzwerke, bauen Schwellwertschalter durch Neurone nach und zeigen, dass selbst einfache Netze dieser Art praktisch jede logische oder auch arithmetische Funktion berechnen können [MP43]. Weiter entstehen erste Computervorläufer („***Elektronengehirne***“), u.a. unterstützt von Konrad Zuse, der es leid war, ballistische Bahnen per Hand zu berechnen.

1947 nennen WALTER PITTS und WARREN MCCULLOCH ein praktisches Anwendungsgebiet (in ihrer Arbeit von 1943 wurde noch kein solches genannt), nämlich die Erkennung räumlicher Muster durch Neuronale Netze [PM47].

1949: DONALD O. HEBB formuliert die klassische *Hebb'sche Lernregel* [Heb49], welche in ihrer allgemeineren Form die Basis fast aller neuronalen Lernverfahren darstellt. Sie besagt, dass die Verbindung zwischen zwei Neuronen verstärkt wird, wenn beide Neurone *gleichzeitig aktiv* sind – die Verstärkung ist also proportional zum Produkt beider Aktivitäten. Hebb konnte diese Regel zwar postulieren, jedoch in Ermangelung neurologischer Forschung nicht verifizieren.

1950 vertritt Neuropsychologe KARL LASHLEY die These, dass die Informationsspeicherung im Gehirn *verteilt* realisiert wird. Begründet wird seine These an Versuchen mit Ratten, bei denen nur der Umfang und nicht der Ort zerstörten Nervengewebes ihre Leistung beeinflusst, aus einem Labyrinth zu finden.

1.2.2 Blütezeit

1951 entwickelt MARVIN MINSKY für seine Dissertation den Neurocomputer ***Snark***, der seine Gewichte[3] bereits automatisch justieren kann, aber nie praktisch eingesetzt wird – da er zwar fleißig rechnet, jedoch niemand so genau weiss, was.

1956 treffen sich auf dem ***Dartmouth Summer Research Project*** renommierte Wissenschaftler und aufstrebende Studierende und diskutieren, salopp gesagt, wie man ein Gehirn nachbilden kann – Unterschiede zwischen Top-Down- und Bottom-Up-Forschung bilden sich heraus. Während die frühen Anhänger der ***Artificial Intelligence*** Fähigkeiten durch Software *nachbilden* möchten, haben die Anhänger der Neuronalen Netze im Sinn, Systemverhalten durch Nachbildung kleinster Systemteile, der Neurone, zu erreichen.

1957 - 1958 entwickeln FRANK ROSENBLATT, CHARLES WIGHTMAN und ihre Mitarbeiter am MIT den ersten erfolgreichen Neurocomputer, das ***Mark I Perceptron*** , welches mit einem 20×20 Pixel großen Bildsensor einfache Ziffern erkennen kann und 512 motorbetriebene Potentiometer besitzt - pro variablem Gewicht eins.

1959 beschreibt FRANK ROSENBLATT verschiedene Varianten des Perceptrons, formuliert und beweist sein *Perceptron-Konvergenz-Theorem.* Er beschreibt an der Retina orientierte Neuronenschichten, Schwellwertschalter und eine Lernregel, welche die Verbindungsgewichte justiert.

3 Wir werden bald erfahren, was Gewichte sind.

1960 stellen BERNARD WIDROW und MARCIAN E. HOFF das ***ADALINE*** (***ADAptive LInear NEuron***) vor [WH60], ein schnell und genau lernendes adaptives System, das wohl das erste verbreitet kommerziell eingesetzte Neuronale Netz darstellte: Es war praktisch in jedem Analogtelefon zur Echtzeit-Echofilterung zu finden und lernte mit der *Widrow-Hoff-Lernregel* bzw. *Deltaregel*. Hoff, ein Mitbegründer von Intel, war zu diesem Zeitpunkt Doktorand von Widrow, der seinerseits einer der Erfinder der modernen Mikroprozessoren war. Einer der Fortschritte der Delta-Regel gegenüber dem ursprünglichen Perceptron-Lernalgorithmus war ihre *Adaptivität*: War man weit von der richtigen Lösung entfernt, so veränderten sich auch die Verbindungsgewichte in größeren Schritten, die in der Nähe des Ziels kleiner werden – Nachteil: bei falscher Anwendung erhält man unendlich kleine Schrittweiten zum Ziel in der Nähe desselben. Während der späteren Flaute und aus Angst vor der wissenschaftlichen Unbeliebtheit der Neuronalen Netze wurde das ADALINE zwischenzeitlich in ***Adaptive Linear Element*** umbenannt – was später wieder rückgängig gemacht wurde.

1961 stellt KARL STEINBUCH Techniken assoziativer Speicherung vor, die als Vorgänger heutiger neuronaler Assoziativspeicher gesehen werden [Ste61]. Er beschreibt weiterhin Konzepte für neuronale Techniken und analysiert ihre Möglichkeiten und Grenzen.

1965 gibt NILS NILSSON in seinem Buch *Learning Machines* einen Überblick über die Fortschritte und Arbeiten dieser Periode der Erforschung Neuronaler Netze. Allgemein nimmt man an, die grundlegenden Prinzipien selbstlernender und damit landläufig gesprochen „intelligenter" Systeme bereits entdeckt zu haben, was aus heutiger Sicht eine maßlose Überschätzung darstellt, aber damals für hohe Popularität und genügend Forschungsmittel sorgte.

1969 veröffentlichen MARVIN MINSKY und SEYMOUR PAPERT eine genaue mathematische Analyse des Perceptrons [MP69] um zu zeigen, dass das Perceptronmodell viele wichtige Probleme gar nicht repräsentieren kann (Stichwörter: *XOR-Problem* und *lineare Separierbarkeit*), und setzen so der Überschätzung, der Popularität und den Forschungsmitteln ein jähes Ende. Die weitergehende Folgerung, dass auch mächtigere Modelle die exakt gleichen Probleme aufweisen, verbunden mit der Prognose, dass das ganze Gebiet ein *research dead-end* sei, bewirken einen fast kompletten Rückgang der Forschungsgelder für die nächsten 15 Jahre, so unzutreffend diese Prognosen aus heutiger Sicht auch waren.

1.2.3 Lange Stille und langsamer Wiederaufbau

Wie oben bereits gesagt – Forschungsgelder wurden extrem knapp. So wurde zwar überall ein wenig weiter geforscht, es gab aber keine Kongresse und sonstigen Veranstaltungen und demzufolge auch wenig Veröffentlichungen. Diese Isolation der einzelnen Forschenden sorgte für die vielen Paradigmen Neuronaler Netze, welche voneinander isoliert entstanden sind: Man forschte, redete aber nicht miteinander.

Trotz der geringen Anerkennung des Gebiets wurden in dieser Zeit die theoretischen Grundlagen für die noch andauernde Renaissance gelegt:

1972 stellt Teuvo Kohonen ein Modell des ***linearen Assoziators***, eines Assoziativspeichermodells vor [Koh72], es wird im gleichen Jahr davon unabhängig von James A. Anderson aus neurophysiologischer Sicht präsentiert [And72].

1973 verwendet Christoph von der Malsburg ein Neuronenmodell, was nichtlinear und biologisch besser motiviert ist [vdM73].

1974 entwickelt Paul Werbos für seine Dissertation in Harvard das *Backpropagation of Error*-Lernverfahren [Wer74], das aber erst ein Jahrzehnt später seine heutige Bedeutung erlangt.

1976 – 1980 und danach werden von Stephen Grossberg viele Arbeiten (z.B. [Gro76]) vorgestellt, in denen eine Vielzahl von neuronalen Modellen mathematisch genau analysiert wird. Er widmet sich weiterhin ausführlich dem Problem, ein Neuronales Netz lernfähig zu halten, ohne bereits erlernte Assoziationen wieder zu zerstören – hieraus entstanden unter Mitarbeit von Gail Carpenter die Modelle der ***Adaptive Resonance Theory***, kurz ART.

1982 beschreibt Teuvo Kohonen die nach ihm benannten ***selbstorganisierenden Karten*** (self organizing feature maps, SOM) [Koh82, Koh98] auf der Suche nach den Mechanismen der Selbstorganisation des Gehirns (er wusste, dass die Informationen über den Aufbau von Wesen im Genom gespeichert sind, das aber ganz wesentlich zu wenig Speicherplatz für eine Struktur wie das Gehirn besitzt – folglich muss sich das Gehirn zum Großteil selbst organisieren und aufbauen).

Weiterhin beschreibt John Hopfield die nach ihm benannten Hopfieldnetze [Hop82], welche durch die Gesetze des Magnetismus in der Physik inspiriert sind. Sie erfuhren wenig technische Anwendungen, aber das Gebiet der Neuronalen Netze kam langsam wieder ins Rollen.

1983 wird von Fukushima, Miyake und Ito das neuronale Modell ***Neocognitron*** zur Erkennung handgeschriebener Zeichen vorgestellt [FMI83], welches eine Erweiterung des schon 1975 entwickelten Cognitrons darstellt.

1.2.4 Renaissance

Durch den Einfluss u.a. John Hopfields, der viele Forscher persönlich von der Wichtigkeit des Gebiets überzeugte, und die weite Publikation von Backpropagation durch Rumelhart, Hinton und Williams machte sich im Gebiet der Neuronalen Netze langsam wieder Aufschwungsstimmung breit.

1985 veröffentlicht John Hopfield einen Artikel, der Wege beschreibt, akzeptable Lösungen für das Travelling Salesman Problem durch *Hopfieldnetze* zu finden.

1986 wird das Lernverfahren *Backpropagation of Error* als Verallgemeinerung der Delta-Regel durch die *Parallel Distributed Processing*-Gruppe separat entwickelt und weit publiziert [RHW86a]: Nicht linear separierbare Probleme wurden durch mehrschichtige Perceptrons lösbar, Marvin Minskys Negativabschätzungen waren mit einem Schlag widerlegt. Weiterhin machte sich zeitgleich in der Artificial Intelligence eine gewisse Ermüdung breit, verursacht durch eine Reihe von Fehlschlägen und untertroffenen Hoffnungen.

Ab dieser Zeit findet eine geradezu explosive Entwicklung des Forschungsgebietes statt, die zwar nicht mehr stichpunktfähig ist, aber von der wir einige Resultate noch wiedersehen werden.

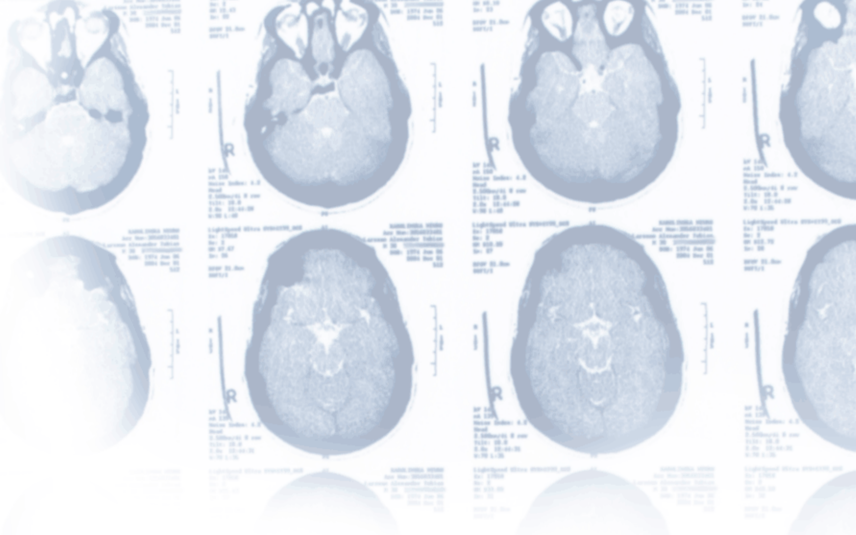
Übungsaufgaben

Aufgabe 1. Suchen Sie für jeden der folgenden Punkte mindestens ein Beispiel:

- ▷ Ein Buch über Neuronale Netze oder Neuroinformatik,
- ▷ eine Arbeitsgruppe an einer Universität, die mit Neuronalen Netzen arbeitet,
- ▷ ein Software-Tool, welches Neuronale Netze realisiert („Simulator“),
- ▷ eine Firma, die Neuronale Netze einsetzt, und
- ▷ ein Produkt oder eine Dienstleistung, die mit Neuronalen Netzen realisiert wurde.

Aufgabe 2. Nennen Sie mindestens vier Anwendungen von technischen Neuronalen Netzen, jeweils zwei aus den Bereichen Mustererkennung und Funktionsapproximation.

Aufgabe 3. Charakterisieren Sie kurz die vier Phasen der Entwicklung Neuronaler Netze und geben Sie aussagekräftige Beispiele für jede Phase an.



Kapitel 2

Biologische Neuronale Netze

Wie lösen biologische Systeme Probleme? Wie funktioniert ein System von Neuronen? Wie kann man dessen Funktionalität erfassen? Wozu sind verschieden große Mengen von Neuronen imstande? Wo im Nervensystem findet Informationsverarbeitung statt? Ein kleiner biologischer Überblick über die große Komplexität einfacher Bausteine Neuronaler Informationsverarbeitung. Zusätzlich die Überleitung zu deren Vereinfachung, um sie technisch adaptieren zu können.

Bevor wir anfangen, Neuronale Netze auf technischer Seite zu beschreiben, ist ein kleiner Exkurs in die Biologie der Neuronalen Netze und der Kognition von Lebewesen sinnvoll – der Leser kann das folgende Kapitel gerne überspringen, ohne etwas Technisches zu verpassen. Ich möchte es aber jedem empfehlen, der ein wenig über die zugrundeliegende Neurophysiologie wissen und sehen möchte, was für eine Karikatur der Natur unsere kleinen Ansätze, die Technischen Neuronalen Netze sind – und wie mächtig die natürlichen Pendants sein müssen, wenn wir mit unseren kleinen Ansätzen schon so weit kommen. Wir wollen nun kurz das Nervensystem von Wirbeltieren betrachten: Wir werden mit sehr grober Granularität beginnen und uns bis ins Gehirn und dann auf die Ebene der Neurone vorarbeiten. Über das ganze Kapitel hinweg haben mir die beiden Bücher [CR00, KSJ00] sehr geholfen, die ich als weiterführende Literatur angeben möchte.

2.1 Das Nervensystem von Wirbeltieren

Das komplette informationsverarbeitende System, das ***Nervensystem*** eines Wirbeltiers, besteht, um erst einmal eine einfache Unterteilung vorzunehmen, aus dem Zentralnervensystem und dem peripheren Nervensystem. In der Realität ist eine ganz so

starre Unterteilung nicht sinnvoll, aber hier hilft sie uns bei der Skizzierung der Informationsverarbeitung im Körper.

2.1.1 Peripheres und zentrales Nervensystem

Das ***periphere Nervensystem*** (***PNS***) besteht aus den Nerven, die außerhalb des Gehirns bzw. Rückenmarks gelegen sind. Diese Nerven bilden ein verzweigtes und sehr dichtes Netz durch den ganzen Körper. Zum peripheren Nervensystem gehören beispielsweise die Spinalnerven, welche vom Rückenmark ausgehen (in Höhe eines jeden Wirbels zwei) und Extremitäten, Hals und Rumpf versorgen, aber auch die direkt zum Gehirn führenden Hirnnerven.

Das ***zentrale Nervensystem*** (***ZNS***) hingegen stellt quasi den „Zentralcomputer“ im Wirbeltier dar. Hier werden Informationen gespeichert und verwaltet, die durch Sinnesorgane von außen aufgenommen werden. Weiter steuert und reguliert es die inneren Vorgänge, und koordiniert nicht zuletzt sämtliche motorischen Leistungen des Organismus. Das zentrale Nervensystem der Wirbeltiere setzt sich zusammen aus dem eigentlichen ***Gehirn*** und dem ***Rückenmark*** (Abb. 2.1 auf der rechten Seite). Wir wollen aber besonderes Augenmerk auf das Gehirn legen. Das Gehirn unterteilt man vereinfachend in vier Bereiche (Abb. 2.2 auf Seite 20), die hier kurz genannt werden sollen.

2.1.2 Das Großhirn ist für abstrakte Denkaufgaben zuständig

Das ***Großhirn*** (***Telencephalon***) ist der Bereich des Gehirns, der sich im Laufe der Evolution mit am meisten verändert hat. Er ist entlang einer Achse, welche den Kopf von Gesichtsseite bis Hinterkopf zentral durchläuft, in zwei Hemisphären geteilt, welche eine in sich gefaltete Struktur aufweisen. Diese Teile sind über einen großen Nervenstrang („***Balken***“)und mehrere kleine miteinander verbunden. Eine Vielzahl von Neuronen liegt in der ca. 2-4 cm dicken ***Großhirnrinde*** (***Cortex***), die in verschiedene ***Rindenfelder*** einzuteilen ist, von denen jedes eine eigene Aufgabe hat. ***Primäre Rindenfelder*** sind hier für die Verarbeitung qualitativer Information zuständig, wie beispielsweise das Verwalten von verschiedenen Wahrnehmungen (z.B. die Verwaltung des Sehsinnes ist Aufgabe des ***visuellen Cortexes***). ***Assoziationsfelder*** hingegen absolvieren höhere, abstraktere Assoziations- und Denkvorgänge; in Ihnen ist auch unser Gedächtnis untergebracht.

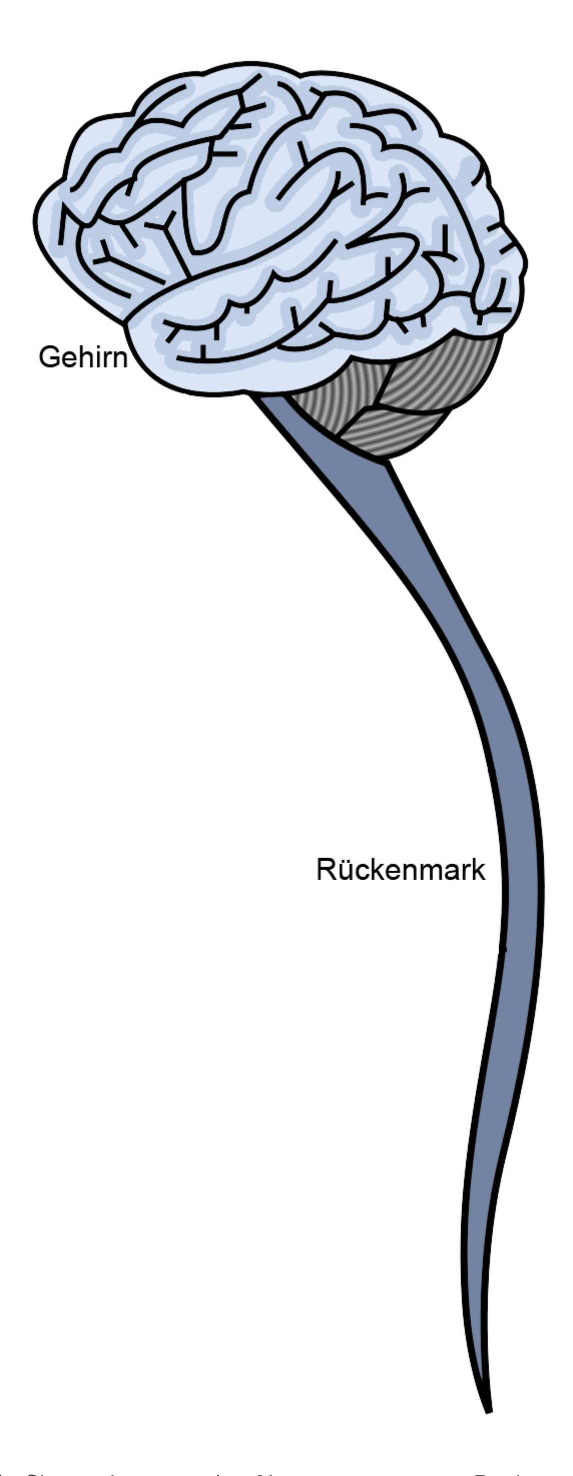


Abbildung 2.1: Skizze des zentralen Nervensystems mit Rückenmark und Gehirn.

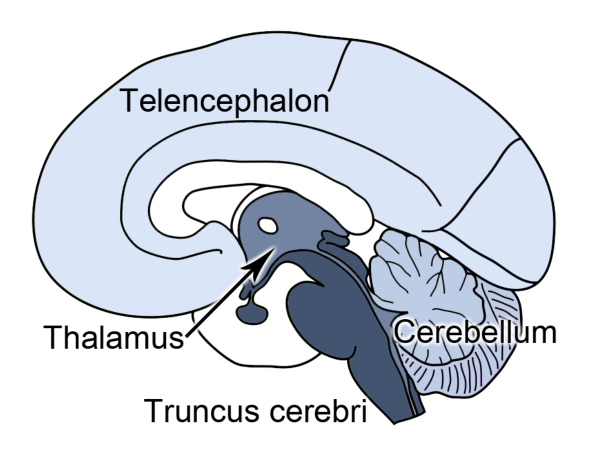


Abbildung 2.2: Skizze des Gehirns. Eingefärbte Bereiche des Gehirns werden im Text behandelt. Je weiter wir von der abstrakten Informationsverarbeitung in Richtung der direkten reflexhaften gehen, desto dunkler sind die Gehirnbereiche eingefärbt.

2.1.3 Das Kleinhirn steuert und koordiniert die Motorik

Das ***Kleinhirn*** (***Cerebellum***) ist unterhalb des Großhirns gelegen, also näher am Rückenmark. Entsprechend dient es weniger abstrakten Funktionen mit höherer Priorität: Hier werden große Teile der motorischen Koordination absolviert, also Gleichgewicht und Bewegungen gesteuert und laufend fehlerkorrigiert. Zu diesem Zweck besitzt das Kleinhirn direkte sensorische Informationen über die Muskellängen sowie akustische und visuelle Informationen. Weiter erhält es auch Meldungen über abstraktere motorische Signale, die vom Großhirn ausgehen.

Beim Menschen ist das Kleinhirn deutlich kleiner als das Großhirn, dies ist aber eher eine Ausnahme: Bei vielen Wirbeltieren ist dieses Verhältnis nicht so stark ausgeprägt. Betrachtet man die Evolution der Wirbeltiere, so ist nicht das Kleinhirn „zu klein“ sondern das Großhirn „zu groß“ geraten (immerhin ist es die am höchsten entwickelte Struktur des Wirbeltiergehirns). Die beiden restlichen Gehirnteile seien ebenfalls noch kurz betrachtet: Das Zwischenhirn und der Hirnstamm.

2.1.4 Das Zwischenhirn steuert grundlegende Körpervorgänge

Das ***Zwischenhirn*** (***Diencephalon***) umfasst wieder einige Teile, von denen nur der ***Thalamus*** einer kurzen Betrachtung unterzogen werden soll: Dieser Teil des Zwi-

schenhirns ist das Medium zwischen sensorischen und motorischen Signalen und Großhirn: Insbesondere wird im Thalamus entschieden, welcher Teil der Information an das Großhirn weitergeleitet wird, so dass gerade weniger wichtige Sinneswahrnehmungen kurzfristig ausgeblendet werden können, um Überlastungen zu vermeiden. Als weiterer Teil des Zwischenhirns steuert der ***Hypothalamus*** eine Vielzahl körperinterner Vorgänge. Das Zwischenhirn ist auch maßgeblich beteiligt am Schlaf-Wach-Rhythmus des Menschen („innere Uhr") und der Schmerzempfindung.

2.1.5 Der Hirnstamm verbindet Hirn und Rückenmark und steuert Reflexe

Verglichen mit dem Zwischenhirn ist der ***Hirnstamm*** bzw. das Stammhirn (***Truncus cerebri***) stammesgeschichtlich deutlich älter: Es markiert, grob gesprochen, das „verlängerte Rückenmark" und damit die Überleitung vom Gehirn zum Rückenmark. Auch der Hirnstamm kann wieder in verschiedene Teile unterteilt werden, von denen einige exemplarisch vorgestellt werden sollen. Die Funktionen gehen weiter vom Abstrakten in Richtung des Grundlegenden. Ein wichtiger Bestandteil ist die ***Pons*** (=Brücke), eine Art Durchgangsstation für sehr viele Nervensignale vom Gehirn an den Körper und umgekehrt.

Wird die Pons geschädigt (beispielsweise durch einen Hirninfarkt), so kann es zum ***Locked-In-Syndrom*** kommen – dem Eingeschlossensein in den eigenen Körper, ein Zustand, in dem es für einen Menschen bei voller geistiger Leistungsfähigkeit völlig unmöglich ist, sich auf irgendeine Weise der Außenwelt zu vermitteln. Man kann nicht sprechen, sich nicht bewegen, während Seh-, Hör-, Geruchs- und Geschmackssinn in aller Regel völlig normal funktionieren. Als letzte Kommunikationsmöglichkeit verbleibt Locked-In-Patienten meist nur die Augenbewegung oder Zwinkern.

Weiter ist der Hirnstamm für viele grundlegende Reflexe zuständig, wie z. B. den reflexartigen Augenlidschluss oder Husten.

Alle Teile des Nervensystems haben eine Sache gemeinsam: Es werden Informationen verarbeitet. Dies geschieht durch riesige Ansammlungen von Milliarden sehr ähnlicher Zellen, die an sich sehr einfach gebaut sind, aber fortlaufend miteinander kommunizieren. Große Gruppen dieser Zellen senden dann koordiniert Signale und erreichen so die gewaltige Informationsverarbeitungskapazität, die wir von unserem Gehirn kennen. Wir wechseln nun von der Ebene der Gehirnteile auf die zelluläre Ebene im Körper – auf die Ebene der Neuronen.

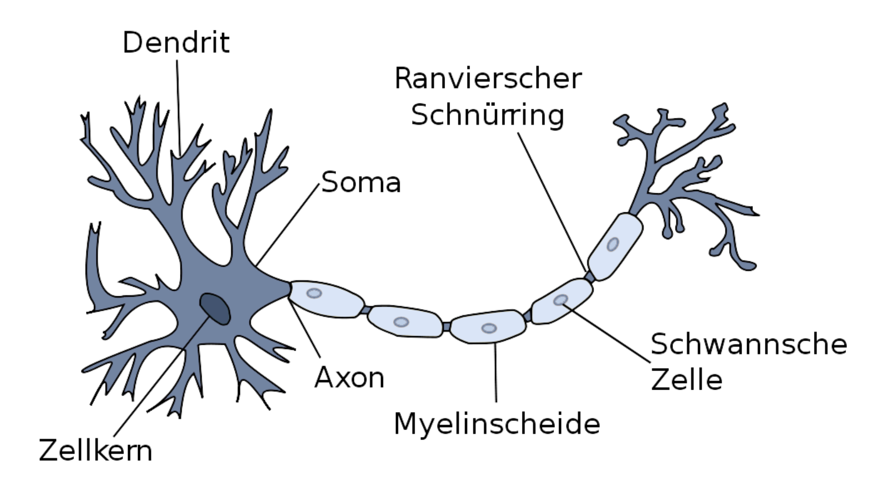


Abbildung 2.3: Skizze eines biologischen Neurons mit Beschriftung von im Text verwendeten Bestandteilen.

2.2 Neuronen sind informationsverarbeitende Zellen

Bevor wir auf die Funktionen und Vorgänge innerhalb eines Neurons genauer eingehen, sei hier zunächst eine Skizze der Neuronenfunktion geliefert: Ein Neuron ist nichts weiter als ein Schalter mit einem Informationseingang und -ausgang. Der Schalter wird aktiviert, wenn genug Reize anderer Neurone am Informationseingang auftreffen. Am Informationsausgang wird dann ein Impuls an z.B. andere Neurone gesendet.

2.2.1 Bestandteile eines Neurons

Wir wollen nun die Einzelbestandteile eines Neurons (Abb. 2.3) betrachten. Wir gehen dabei den Weg, den die elektrische Information im Neuron nimmt. Den Dendriten eines Neurons zugeleitet wird die Information über spezielle Übergangsstellen, die Synapsen.

2.2.1.1 Synapsen gewichten die einzelnen Informationsanteile

Von anderen Neuronen oder sonstigen Zellen eingehende Signale werden einem Neuron über spezielle Übergangsstellen, die ***Synapsen*** zugeleitet. Solch eine Übergangsstelle

liegt meistens an den Dendriten eines Neurons, manchmal auch direkt am Soma. Man unterscheidet elektrische und chemische Synapsen.

Die einfachere von beiden Varianten ist die ***elektrische Synapse*** . Ein elektrisches Signal, welches auf der Synapse eingeht, also von der *präsynaptischen* Seite kommt, wird direkt in den *postsynaptischen* Zellkern fortgeleitet. Es liegt also eine direkte, starke, nicht regulierbare Verbindung von Signalgeber zu Signalempfänger vor, zum Beispiel sinnvoll für Fluchtreflexe, die in einem Lebewesen „hart codiert" sein müssen.

Die ausgeprägtere Variante ist die ***chemische Synapse***. Hier findet keine direkte elektrische Kopplung von Quelle und Ziel statt, sondern diese Kopplung ist unterbrochen durch den ***synaptischen Spalt***. Dieser Spalt trennt die prä- und postsynaptische Seite elektrisch voneinander. Dennoch muss ja Information fließen, werden Sie jetzt denken, also wollen wir darauf eingehen, wie das passiert: Nämlich nicht elektrisch, sondern *chemisch*. Auf der präsynaptischen Seite des synaptischen Spalts wird das elektrische Signal in ein chemisches konvertiert, indem dort chemische Signalstoffe freigesetzt werden (sog. ***Neurotransmitter***). Diese Neurotransmitter überwinden den synaptischen Spalt und übertragen die Information in den Zellkern (das ist sehr vereinfacht ausgedrückt, wir werden später noch sehen, wie das genau funktioniert), wo sie wieder in elektrische Information umgewandelt wird. Die Neurotransmitter werden sehr schnell wieder abgebaut, so dass auch hier sehr genaue Informations-Impulse möglich sind.

Trotz der viel komplizierteren Funktionsweise hat die chemische Synapse im Vergleich zur elekrischen Variante eklatante Vorteile:

Einwegschaltung: Die chemische Synapse ist eine Einwegschaltung. Dadurch, dass der prä- und postsynaptische Teil nicht direkt elektrisch verbunden sind, können elektrische Impulse im postsynaptischen Teil nicht auf den präsynaptischen überschlagen.

Regulierbarkeit: Es gibt eine Vielzahl verschiedener Neurotransmitter, die noch dazu in verschiedenen Mengen in einem synaptischen Spalt freigesetzt werden können. So gibt es Neurotransmitter, die anregend auf den postsynaptischen Zellkern wirken, aber auch andere, die eine solche Anregung wieder abflauen lassen. Manche Synapsen geben ein stark anregendes Signal weiter, manche nur schwach anregende Signale. Die Regulierungsvielfalt ist enorm, und dass die Synapsen hier auch noch variabel sind, also mit der Zeit eine stärkere oder schwächere Verbindung darstellen können, ist einer der zentralen Punkte bei Betrachtung der Lernfähigkeit des Gehirns.

2.2.1.2 Dendriten sammeln alle Informationsanteile

Dendriten verästeln sich baumartig vom Zellkern des Neurons (den man *Soma* nennt) und dienen der Aufnahme von elektrischen Signalen aus vielen verschiedenen Quellen, die dann in den Zellkern übertragen werden. Die sich verästelnde Menge von Dendriten wird auch ***Dendritenbaum*** genannt.

2.2.1.3 Im Soma werden die gewichteten Informationsanteile aufkumuliert

Nachdem über Synapsen und Dendriten eine Fülle an aktivierenden (=anregenden) und inhibierenden (=abschwächenden) Signalen beim Zellkern (***Soma***) eingetroffen ist, kumuliert das Soma diese Signale auf. Sobald das aufkumulierte Signal einen gewissen Wert (Schwellwert genannt) überschreitet, löst der Neuronenzellkern seinerseits einen elektrischen Impuls aus, der dann zur Weiterleitung an die nachfolgenden Neuronen bestimmt ist, zu denen das aktuelle Neuron verbunden ist.

2.2.1.4 Das Axon leitet ausgehende Impulse weiter

Die Weiterleitung des Impulses zu anderen Neuronen erfolgt durch das ***Axon***. Das Axon ist ein fadenartiger Fortsatz des Somas. Ein Axon kann im Extremfall ca. einen Meter lang werden (z.B. im Rückenmark). Das Axon ist elektrisch isoliert, um das elektrische Signal besser leiten zu können (später mehr dazu) und mündet in Dendriten, um die Information an z.B. andere Neurone weiterzugeben. Wir sind also wieder am Anfang unserer Beschreibung der Neuronenbestandteile angelangt. Natürlich kann ein Axon aber auch Informationen an andere Zellenarten zu deren Steuerung übertragen.

2.2.2 Elektrochemische Vorgänge im Neuron und seinen Bestandteilen

Nachdem wir nun den Weg eines elektrischen Signals von den Dendriten über die Synapsen in den Zellkern, und von dort über das Axon in weitere Dendriten verfolgt haben, wollen wir einen kleinen Schritt von der Biologie in Richtung Technik gehen. Auf diesem Weg soll vereinfacht vorgestellt werden, wie die Informationsverarbeitung elektrochemisch vonstatten geht.

2.2.2.1 Neuronen erhalten ein elektrisches Membranpotential aufrecht

Ein grundlegender Aspekt ist, dass die Neurone gegenüber ihrer Umwelt eine elektrische Ladungsdifferenz, ein *Potential* aufweisen. Innerhalb der ***Membran*** (=Hülle) des Neurons herrscht also eine andere Ladung vor als außen. Diese Ladungsdifferenz ist ein zentraler Begriff, den man braucht, um die Vorgänge im Neuron zu verstehen, wir nennen sie ***Membranpotential***. Das Membranpotential, also der Ladungsunterschied, entsteht durch mehrere Arten geladener Atome (***Ione***), die innerhalb und außerhalb des Neurons unterschiedlich hoch konzentriert sind. Wenn wir von innen nach außen durch die Membran stoßen, werden wir bestimmte Arten Ione häufiger oder weniger häufig vorfinden als innen, wir nennen diesen Abfall oder Anstieg der Konzentration einen ***Konzentrationsgradienten***.

Betrachten wir das Membranpotential zunächst für den Ruhezustand des Neurons, nehmen wir also an, es treffen gerade keine elektrischen Signale von außen ein. In diesem Fall beträgt das Membranpotential -70 mV. Da wir gelernt haben, dass dieses Potential von Konzentrationsgradienten verschiedener Ionen abhängt, ist natürlich eine zentrale Frage, wie diese Konzentrationsgradienten aufrecht erhalten werden: Normalerweise herrscht ja überall Diffusion vor, also sind alle Ionen bestrebt, Konzentrationsgefälle abzubauen und sich überall gleichmäßig zu verteilen. Würde das passieren, würde das Membranpotential gegen 0 mV gehen, schlussendlich würde also kein Membranpotential mehr vorhanden sein. Das Neuron erhält sein Membranpotential also aktiv aufrecht, um Informationsverarbeitung betreiben zu können. Wie geht das vonstatten?

Das Geheimnis liegt in der Membran selbst, die für manche Ione durchlässig ist, für andere aber nicht. Um das Potential aufrecht zu erhalten, wirken hier mehrere Mechanismen gleichzeitig:

Konzentrationsgradient: Wie schon beschrieben, versuchen die Ionen, immer möglichst gleichverteilt vertreten zu sein. Ist innerhalb des Neurons die Konzentration eines Ions höher als außen, versucht es nach außen zu diffundieren und umgekehrt. Das positiv geladene Ion K^+ (Kalium) ist im Neuron häufig, außerhalb des Neurons weniger anzutreffen, und diffundiert darum langsam durch die Membran aus dem Neuron hinaus. Eine weitere Sammlung negativer Ionen, zusammenfassend A^- genannt, bleibt aber im Neuron, da die Membran hierfür nicht durchlässig ist. Das Neuroneninnere wird also negativ: Negative A-Ionen bleiben, positive K-Ionen verschwinden, das Innere der Zelle wird negativer. Dies führt uns zu einem weiteren Gradienten.

Elektrischer Gradient: Der elektrische Gradient wirkt dem Konzentrationsgradienten entgegen. Das Zellinnere ist mittlerweile sehr negativ geworden, also zieht es positive Ionen an: K^+ möchte nun wieder in die Zelle hinein.

Würde man diese beiden Gradienten nun einfach sich selbst überlassen, so würden sie sich irgendwann ausgleichen, einen stabilen Zustand erreichen und ein Membranpotential von -85 mV würde entstehen. Wir wollen aber auf ein Ruhemembranpotential von -70 mV hinaus, es muss also Störkomponenten geben, die dies verhindern. Zum einen gibt es noch ein weiteres wichtiges Ion, Na^+ (Natrium), für das die Membran zwar nicht besonders durchlässig ist, das aber trotzdem langsam durch die Membran in die Zelle einströmt. Das Natrium fühlt sich hierbei doppelt nach innen getrieben: Zum einen gibt es weniger Natrium im inneren des Neurons als außen, zum anderen ist Natrium positiv, das Zellinnere aber negativ, ein zweiter Grund, in die Zelle zu wollen.

Durch die geringe Natriumdiffusion ins Zellinnere nimmt die Natriumkonzentration im Zellinneren zu, gleichzeitig wird das Zellinnere aber weniger negativ, so dass der Einstrom von K^+ langsamer wird (wir sehen: Das ist ein komplizierter Mechanismus, in dem alles sich gegenseitig beeinflusst). Durch Natrium wird das Zellinnere tendentiell weniger negativ gegenüber der Umwelt. Auch mit diesen beiden Ionen könnte aber immer noch ein Stillstand erreicht werden, in dem sich alle Gradienten ausgleichen und nichts mehr passiert. Nun kommt aber das fehlende Mosaiksteinchen, auf das wir warten: Eine „Pumpe“ (eigentlich das Protein ***ATP***) bewegt aktiv Ionen entgegen der Richtung, zu der sie eigentlich möchten!

Natrium wird aktiv aus der Zelle rausgepumpt, obwohl es entlang des Konzentrations- und des elektrischen Gradienten in die Zelle möchte.

Kalium hingegen diffundiert stark aus der Zelle heraus, wird aber wieder aktiv hineingepumpt.

Aus diesem Grund nennen wir die Pumpe auch ***Natrium-Kalium-Pumpe***. Die Pumpe erhält sowohl für Natrium als auch für Kalium den Konzentrationsgradienten aufrecht, so dass eine Art Fließgleichgewicht entsteht und das Ruhepotential schlussendlich bei den beobachteten -70 mV landet. Zusammenfassend wird das Membranpotential also aufrecht erhalten, indem die Membran für manche Ionen gar nicht durchlässig ist und andere Ionen aktiv entgegen der Konzentrations- und elektrischen Gradienten gepumpt werden. Nachdem wir nun wissen, dass jedem Neuron ein Membranpotential zueigen ist, wollen wir genau betrachten, wie ein Neuron Signale empfängt und versendet.

2.2.2.2 Veränderungen im Membranpotential aktivieren das Neuron

Oben haben wir gelernt, dass Natrium und Kalium durch die Membran hindurchdiffundieren können, Natrium langsam, Kalium schneller. Dies geschieht durch in der Membran enthaltene Kanäle, Natrium- bzw. Kaliumkanäle. Neben diesen immer geöffneten Kanälen, die für die Diffusion verantwortlich sind und durch die Natrium-Kalium-Pumpe ausgeglichen werden, gibt es auch Kanäle, die nicht immer geöffnet sind, sondern nur „nach Bedarf“ geöffnet werden. Da sich durch eine Öffnung dieser Kanäle die Konzentration von Ionen innerhalb und außerhalb der Membran verändern, ändert sich logischerweise auch das Membranpotential.

Diese steuerbaren Kanäle werden geöffnet, wenn der aufkumulierte eingehende Reiz einen gewissen Schwellwert überschreitet. Eingehende Reize können beispielsweise von anderen Neuronen kommen oder aber von anderen Ursachen herrühren: So gibt es zum Beispiel spezialisierte Formen von Neuronen, Sinneszellen, für die z.B. Lichteinfall einen solchen Reiz darstellen kann. Fällt dann genug Licht ein, um den Schwellwert zu überschreiten, werden steuerbare Kanäle geöffnet.

Der besagte Schwellwert (das ***Schwellenwertpotential***) liegt bei ca. -55 mV. Sobald dieses von den eingehenden Reizen erreicht wird, wird das Neuron aktiviert und ein elektrisches Signal, ein ***Aktionspotential*** wird ausgelöst. Dieses Signal wird dann weitergeleitet an diejenigen Zellen, die mit dem betrachteten Neuron verbunden sind, also ihm „zuhören“. Die Phasen des Aktionspotentials wollen wir etwas genauer betrachten (Abb. 2.4 auf der folgenden Seite):

Ruhezustand: Nur die immer geöffneten Kalium- und Natriumkanäle liegen offen, das Membranpotential liegt bei -70 mV und wird vom Neuron aktiv dort gehalten.

Stimulus bis Schwellwert: Ein Reiz (***Stimulus***) öffnet Kanäle, so dass Natrium einströmen kann. Die Ladung des Zellinneren wird positiver. Sobald das Membranpotential über den Schwellenwert von -55 mV geht, wird das Aktionspotential ausgelöst, indem sehr viele Natriumkanäle geöffnet werden.

Depolarisation: Natrium strömt ein. Wir erinnern uns, dass Natrium einströmen möchte sowohl, weil davon deutlich weniger in der Zelle vorhanden ist als außen, als auch, weil in der Zelle ein negatives Milieu vorherrscht, durch das das positive Natrium angezogen wird. Durch den starken Einstrom steigt das Membranpotential drastisch an, bis auf ca. $+30$ mV. Dies ist dann der elektrische Impuls, das Aktionspotential.

Repolarisation: Natriumkanäle werden nun geschlossen, dafür werden die Kaliumkanäle geöffnet. Das positiv geladene Kalium will nun aus dem ebenfalls positiven

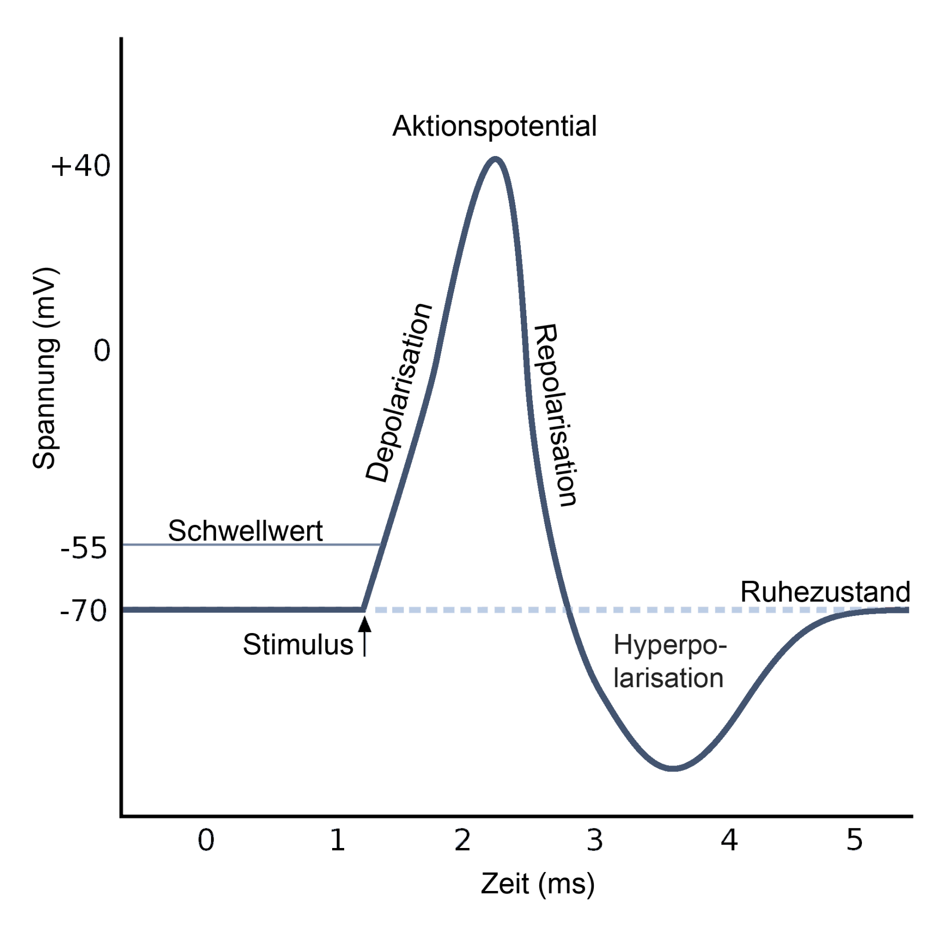


Abbildung 2.4: Auslösung eines Aktionspotentials über die Zeit.

Zellinneren heraus. Zusätzlich ist es im Zellinneren wesentlich höher konzentriert als außen, was den Ausstrom noch beschleunigt. Das Zellinnere wird wieder negativer geladen als die Außenwelt.

Hyperpolarisation: Sowohl Natrium- als auch Kaliumkanäle werden wieder geschlossen. Zunächst ist das Membranpotential nun leicht negativer als das Ruhepotential, was daher kommt, dass die Kaliumkanäle sich etwas träger schließen, was Kalium (positiv geladen) aufgrund seiner geringeren Konzentration außerhalb der Zelle ausströmen lässt. Nach einer ***Refraktärzeit*** von $1-2$ ms ist dann der Ruhezustand wiederhergestellt, so dass das Neuron auf neue Reize wieder mit einem Aktionspotential reagieren kann. Die Refraktärzeit ist, einfach ausgedrückt, eine Zwangspause, welche ein Neuron einhalten muss, um sich zu regenerieren. Je kürzer sie ist, desto öfter kann ein Neuron pro Zeit feuern.

Der so entstandene Impuls wird dann durch das Axon fortgeleitet.

2.2.2.3 Im Axon wird ein Impuls auf saltatorische Weise weitergeleitet

Wir haben schon gelernt, dass das ***Axon*** zur Fortleitung des Aktionspotentials über lange Distanzen dient (zur Erinnerung: Eine Illustration eines Neurons inklusive Axon findet sich in Abb. 2.3 auf Seite 22). Das Axon ist ein fadenartiger Fortsatz des Somas. Bei Wirbeltieren ist es in der Regel von einer ***Myelinscheide*** umgeben, welche aus ***Schwannschen Zellen*** (im PNS) oder ***Oligodendrozyten*** (im ZNS) besteht[1], die das Axon elektrisch sehr gut isolieren. Zwischen diesen Zellen befinden sich im Abstand von $0.1-2$mm Lücken, die sogenannten ***Ranvierschen Schnürringe***, die jeweils dort auftreten, wo eine Isolationszelle zuende ist und die nächste anfängt. An einem solchen Schnürring ist das Axon logischerweise schlechter isoliert.

Man mag nun annehmen, dass diese schlecht isolierten Schnürringe von Nachteil für das Axon sind: Dem ist aber nicht so. An den Schnürringen ist ein Stoffaustausch zwischen intrazellulärem und extrazellulärem Raum möglich, der an Teilen des Axons, welche zwischen zwei Schnürringen liegen (***Internodien***) und so durch die Myelinscheide isoliert sind, nicht funktioniert. Dieser Stoffaustausch ermöglicht die Erzeugung von Signalen, ganz ähnlich der Erzeugung des Aktionspotentials im Soma. Die Weiterleitung eines Aktionspotentials funktioniert nun folgendermaßen: Es läuft nicht kontinuierlich entlang des Axons, sondern springt von einem Schnürring zum nächsten. Es läuft also eine Reihe von Depolarisationen entlang der Ranvierschnürringe. Ein Aktionspotential

1 Sowohl Schwannsche Zellen als auch Oligodendrozyten sind Ausprägungen der ***Gliazellen***, von denen es ca. 50 mal mehr gibt als Neuronen, und die die Neuronen umgeben (Glia = Leim), voneinander isolieren, mit Energie versorgen, etc.

löst dominoartig das nächste aus, meist sind hierbei sogar mehrere Schnürringe gleichzeitig aktiv. Durch das „Springen“ des Impulses von einem Schnürring zum nächsten kommt diese Impulsleitung auch zu ihrem Namen: ***Saltatorische Impulsleitung***.

Es ist offensichtlich, dass der Impuls schneller voran kommt, wenn seine Sprünge größer sind. So wird bei Axonen mit großen Internodien (2 mm) eine Signalausbreitungsgeschwindigkeit von ca. 180 Metern pro Sekunde erreicht. Die Internodien können aber nicht beliebig groß werden, da das weiterzuleitende Aktionspotential sonst bis zum nächsten Schnürring zu sehr verblassen würde. Die Schnürringe haben also auch die Aufgabe, das Signal regelmäßig zu verstärken. Am Ende des Axons hängen dann – oft über Dendriten und Synapsen verbunden – die Zellen, welche das Aktionspotenzial empfangen. Wie oben schon angedeutet, können Aktionspotentiale aber nicht nur durch über die Dendriten eingehende Information von anderen Neuronen entstehen.

2.3 Rezeptorzellen sind abgewandelte Neurone

Aktionspotentiale können auch durch sensorische Informationen, die ein Lebewesen aus seiner Umwelt mittels Sinneszellen aufnimmt, herrühren. Spezialisierte ***Rezeptorzellen*** können für sie spezifische Reizenergien wie Licht, Temperatur und Schall oder das Vorhandensein bestimmter Moleküle wahrnehmen (wie es z.B. der Geruchssinn tut). Dies funktioniert, da diese Sinneszellen eigentlich abgewandelte Neurone sind: Hier werden keine elektrischen Signale über Dendriten aufgenommen, sondern das Vorhandensein des für die Rezeptorzelle spezifischen Reizes sorgt dafür, dass sich Ionenkanäle öffnen und ein Aktionspotential ausgebildet wird. Dieser Vorgang des Umwandelns von Reizenergie in Veränderungen im Membranpotential nennt sich ***Sensorische Transduktion***. In aller Regel ist die Reizenergie selbst zu schwach, um direkt Nervensignale auszulösen, und so findet entweder während der Transduktion oder durch den ***Reizleitenden Apparat*** auch gleich eine Signalverstärkung statt. Das resultierende Aktionspotential kann von anderen Neuronen verarbeitet werden und gelangt dann in den Thalamus, der, wie wir schon gelernt haben, als Tor zum Cerebralen Cortex Sinneseindrücke nach der momentanen Relevanz aussortieren und so einen Überfluss an zu verwaltenden Informationen verhindern kann.

2.3.1 Es existieren verschiedenste Rezeptorzellen für viele Arten von Wahrnehmungen

Primärrezeptoren senden ihre Impulse direkt ans Nervenssystem. Schmerzempfindung ist hierfür ein gutes Beispiel. Hier ist die Reizstärke proportional zur Amplitude

des Aktionspotentials, technisch ausgedrückt findet hier also eine Amplitudenmodulation statt.

Sekundärrezeptoren senden hingegen durchgehend Impulse. Diese Impulse steuern dann die Menge des zugehörigen Neurotransmitters zur Weitergabe des Reizes, der wiederum die Frequenz der Aktionspotentiale des empfangenden Neurons steuert. Hier handelt es sich um Frequenzmodulation, eine Kodierung des Reizes, bei der man Zu- und Abnahme eines Reizes besser wahrnehmen kann.

Rezeptorzellen können einzeln vorkommen oder aber komplexe Sinnesorgane bilden (z.B. Augen oder Ohren). Es können sowohl Reize im Körper empfangen werden (das machen dann ***Enterorezeptoren***) wie auch Reize außerhalb des Körpers (hierfür sind die ***Exterorezeptoren*** zuständig).

Nachdem wir nun skizziert haben, wie Information aus der Umwelt aufgenommen wird, ist interessant zu betrachten, auf welche Weise sie *verarbeitet* wird.

2.3.2 Informationsverarbeitung findet auf jeder Ebene des Nervensystems statt

Es ist nämlich keinesfalls so, dass alle Informationen ins Gehirn geleitet werden, dort verarbeitet werden, und das Gehirn danach für eine „Ausgabe“ in Form von motorischen Impulsen sorgt (das einzige, was ein Lebewesen in seiner Umwelt wirklich tun kann, ist ja, sich zu bewegen). Die Informationsverarbeitung ist komplett dezentral angelegt. Um das Prinzip zu verdeutlichen, wollen wir kurz ein paar Beispiele betrachten, für die wir in unserer Informationsverarbeitungshierarchie wieder vom Abstrakten zum Grundlegenden wandern.

- Dass im Großhirn als höchstentwickelter Informationsverarbeitender Struktur in der Natur Informationsverarbeitung stattfindet, ist klar.

- In der Hierarchie deutlich tiefer liegt das Mittelhirn und der Thalamus, den wir als Tor zur Großhirnrinde schon kennengelernt haben: Auch das von ihm betriebene Filtern von Informationen nach aktueller Relevanz ist eine sehr wichtige Art der Informationsverarbeitung. Doch auch der Thalamus erhält keine Reize von außen, die nicht bereits vorverarbeitet wurden. Machen wir einen Sprung zur untersten Ebene, den Sinneszellen.

- Bereits auf der untersten Ebene, den Rezeptorzellen, wird Information nicht nur aufgenommen und weitergeleitet, sondern auch direkt verarbeitet. Einer der Hauptaspekte zu diesem Thema ist die Verhinderung von „Dauerreizen“ an das

Zentralnervensystem durch ***sensorische Adaption***: Bei kontinuierlicher Reizung werden sehr viele Rezeptorzellen automatisch unempfindlicher. Rezeptorzellen stellen also keine direkte Abbildung von spezifischer Reizenergie auf Aktionspotentiale da, sondern sind abhängig von der Vergangenheit. Weitere Sensoren ändern die Empfindlichkeit je nach Situation: Es gibt Geschmacksrezeptoren, die je nach Ernährungszustand des Organismus mehr oder weniger stark auf denselben Reiz ansprechen.

▷ Selbst *bevor* ein Reiz zu den Rezeptorzellen vorstößt, kann durch einen vorgeschalteten signalführenden Apparat schon Informationsverarbeitung stattfinden, beispielsweise in Form von Verstärkung: Die Ohrmuschel und das Innenohr haben eine spezifische schallverstärkende Form, welche es – in Verbindung mit den Sinneszellen des Hörsinns – ebenfalls ermöglich, dass der Nervenreiz nur *logarithmisch* mit der Intensität des gehörten Signals ansteigt. Dies ist bei näherer Betrachtung auch sehr notwendig, da der Schalldruck der Signale, für die das Ohr gemacht ist, über viele Zehnerpotenzen variieren kann. Eine logarithmische Messweise ist hier von Vorteil. Erstens wird Überlastung vermieden, und dass zweitens die Intensitätsmessung bei intensiven Signalen dadurch weniger genau wird, macht auch nichts: Wenn neben einem gerade ein Düsenjet startet, sind winzige Schwankungen im Lärmpegel zu vernachlässigen.

Um noch etwas mehr Bauchgefühl über Sinnesorgane und Informationsverarbeitung im Organismus zu erhalten, sollen nun kurz „gängige“, also in der Natur häufig anzutreffende Lichtsinnesorgane beschrieben werden. Beim dritten beschriebenen Lichtsinnesorgan, dem Einzellinsenauge, gehen wir dann auf Informationsverarbeitung ein, die noch direkt im Auge stattfindet.

2.3.3 Eine Skizze häufiger Lichtsinnesorgane

Für viele Lebewesen hat es sich als extrem nützlich erwiesen, elektromagnetische Strahlung in bestimmten Bereichen wahrzunehmen. Konsequenterweise sind Sinnesorgane entstanden, welche solch elektromagnetische Strahlung feststellen können, und der Wellenlängenbereich dieser Strahlung, welcher für den Menschen wahrnehmbar ist, heißt dadurch *sichtbarer Bereich* oder schlicht *Licht*. Verschiedene Wellenlängen dieser sichtbaren Strahlung nehmen wir Menschen durch verschiedene Farben wahr. Der sichtbare Bereich der elektromagnetischen Strahlung ist nicht bei allen Lebewesen gleich, manche Lebewesen können Farben (=Wellenlängenbereiche) nicht sehen, die wir sehen können, andere Lebewesen können sogar zusätzliche Wellenlängenbereiche (z.B. im UV-Bereich) wahrnehmen. Bevor wir zum Menschen kommen, wollen wir kurz – um etwas breiter



Abbildung 2.5: Facettenaugen einer Raubfliege

angelegtes Wissen zum Sehsinn zu erhalten – zwei Sehsinnesorgane betrachten, die evolutionär gesehen vor dem Menschen da waren.

2.3.3.1 Komplexaugen und Lochkameraaugen bieten nur zeitlich bzw. örtlich hohe Auflösung

Betrachten wir als erstes das sogenannte ***Komplexauge*** (Abb. 2.5), auch ***Facettenauge*** genannt, welches beispielsweise bei Insekten und Krustentieren vorkommt. Das Komplexauge besteht aus vielen kleinen, separaten ***Einzelaugen***, die, wenn man das Komplexauge von außen betrachtet, auch deutlich sichtbar sind und ein wabenähnliches Muster erzeugen. Jedes Einzelauge im Komplexauge verfügt über eine eigene Nervenfaser als Anbindung an das Insektengehirn. Da wir die Einzelaugen sehen können, ist klar, dass die Anzahl der Bildpunkte, also die spatiale Auflösung bei Komplexaugen recht niedrig liegen muss: Das Bild ist unscharf. Komplexaugen bieten aber auch Vorteile, gerade für schnellfliegende Insekten: Bestimmte Arten Komplexaugen verarbeiten mehr als 300 Bilder pro Sekunde (dem Menschen hingegen erscheinen schon Kinofilme mit 25 Bildern pro Sekunde als flüssige Bewegung).

Lochkameraaugen kommen zum Beispiel bei Krakenarten vor und funktionieren – man errät es – ähnlich einer Lochkamera. Es gibt ein winziges Lichteintrittsloch, welches ein scharfes Bild auf dahinterliegende Sinneszellen projiziert, die räumliche Auflösung ist hier also deutlich höher als beim Komplexauge. Aufgrund des winzigen Eintrittsloches ist das Bild aber sehr lichtschwach.

2.3.3.2 Einzellinsenaugen kombinieren beide Stärken, sind aber komplexer aufgebaut

Die bei Wirbeltieren verbreitete Art des Lichtsinnesorgans ist das ***Einzellinsenauge***: Es vermittelt ein scharfes, hochaufgelöstes Bild der Umwelt, bei hoher bzw. variabler Lichtstärke. Dafür ist es komplizierter aufgebaut. Wie beim Lochkameraauge fällt Licht durch eine Öffnung ein (***Pupille***) und wird im Auge auf eine Schicht Sinneszellen projiziert (***Netzhaut*** oder ***Retina***). Im Unterschied zum Lochkameraauge kann der Öffnungsgrad der Pupille allerdings den Helligkeitsverhältnissen angepasst werden (dies geschieht durch den ***Iris***-Muskel, der die Pupille vergrößert oder verkleinert). Diese Unterschiede im Pupillenöffnungsgrad machen eine aktive Scharfstellung des Bildes notwendig, weswegen das Einzellinsenauge noch eine ebenfalls verstellbare ***Linse*** enthält.

2.3.3.3 Die Retina ist nicht nur Empfänger, sondern verarbeitet Informationen

Die auftreffenden Lichtsignale werden in der Retina aufgenommen und direkt durch mehrere Schichten informationsverarbeitender Zellen vorverarbeitet. Wir wollen kurz verschiedene Punkte dieser Informationsvorverarbeitung betrachten und gehen dabei den Weg, den die vom Licht eingebrachte Information nimmt:

Photorezeptoren empfangen das Lichtsignal und lösen Aktionspotentiale aus (es gibt verschiedene Rezeptoren für verschiedene Farbanteile und Lichtintensitäten). Diese Rezeptoren sind der eigentlich lichtempfangende Teil der Retina und derart empfindlich, dass bereits das Auftreffen von einem einzigen Photon ein Aktionspotential auslösen kann. Mehrere Photorezeptoren leiten dann ihre Signale an eine einzige

Bipolarzelle weiter, hier findet also bereits eine Zusammenfassung der Information statt. Schlussendlich gelangt das nun umgewandelte Lichtsignal wieder von jeweils mehreren Bipolarzellen[2] in die

Ganglienzellen. Es können verschieden viele Bipolarzellen ihre Information an eine Ganglienzelle weiterleiten. Je höher die Zahl der Photorezeptoren, von denen die Ganglienzelle betroffen ist, um so größer ist der Wahrnehmungsbereich, das ***Rezeptive Feld***, welches die Ganglien abdeckt – und um so weniger Bildschärfe ist im Bereich dieser Ganglienzelle gegeben. Hier werden also direkt in der Retina bereits Informationen aussortiert und das Gesamtbild z.B. in Randsichtbereichen verunschärft. Bis jetzt haben wir die Informationsverarbeitung in der Retina aber

2 Es gibt wieder verschiedene Arten Bipolarzellen, deren Betrachtung hier aber zu weit führen würde.

als reine Top-Down-Struktur kennengelernt. Dies bringt uns nun zur Betrachtung der

Horizontal- und Amakrinzellen. Diese Zellen sind nicht von vorne nach hinten, sondern lateral verbunden und machen es möglich, dass Lichtsignale sich direkt während der Informationsverarbeitung in der Retina *lateral* beeinflussen – eine viel mächtigere Art der Informationsverarbeitung als komprimieren und verunschärfen. Horizontalzellen ist es hierbei möglich, von einem Photorezeptor angeregt, andere nahe Photorezeptoren ebenfalls anzuregen und gleichzeitig weiter entfernte Bipolarzellen und Rezeptoren zu hemmen. Dies sorgt für die klare Wahrnehmung von Konturen und hellen Punkten. Amakrinzellen können weiter bestimmte Reize verstärken, indem sie Informationen von Bipolarzellen auf mehrere Ganglienzellen verteilen oder auch Ganglien hemmen.

Wir sehen an diesem Anfang des Wegs visueller Information ins Hirn, dass Informationsverarbeitung zum einen vom ersten Moment der Informationsaufnahme stattfindet – und zum anderen parallel in Millionen von informationsverarbeitenden Zellen stattfindet. Kein Systembestandteil ruht jemals, und aus dieser massiven Verteilung der Arbeit zieht das Nervensystem seine Macht und Fehlerresistenz.

2.4 Neuronenmengen in unterschiedlich hoch entwickelten Lebewesen

Hier nun ein Überblick, welche verschiedenen Lebewesen was für eine Kapazität an Neuronen besitzen (zum Großteil aus [RD05]):

302 Neuronen benötigt das Nervensystem eines *Fadenwurms*, der einen beliebten Modellorganismus in der Biologie darstellt. Fadenwürmer leben im Erdboden und ernähren sich von Bakterien.

10^4 Neuronen ergeben eine *Ameise* (der Einfachheit halber sei vernachlässigt, dass manche Ameisenarten auch mehr oder weniger leistungsfähige Nervensysteme aufweisen können). Durch verschiedene Lockstoffe und Düfte sind Ameisen zu komplexem Sozialverhalten und zur Bildung von riesigen Staaten mit Millionen von Individuen in der Lage. Sieht man nicht die Ameise, sondern einen solchen Staat als Individuum an, so kommt man ungefähr auf die kognitive Leistungsfähigkeit eines Schimpansen bis hin zum Menschen.

Mit 10^5 Neuronen können wir das Nervensystem einer *Fliege* bauen. Eine Fliege kann im dreidimensionalen Raum in Echtzeit ausweichen, durch einen Looping an der

Decke landen, besitzt umfangreiche Sensorik durch Facettenaugen, Tasthaare, Nerven an den Beinenden und vieles mehr. Eine Fliege hat also „in Hardware“ umfangreiche Differential- und Integralrechnungen in hohen Dimensionen implementiert. Wir alle wissen, dass eine Fliege schwer zu fangen ist. Natürlich werden auch die Körperfunktionen von Neuronen gesteuert, diese seien aber hier außen vor gelassen.

Mit $0.8 \cdot 10^6$ Neuronen haben wir genug Gehirnmasse für eine *Honigbiene*. Honigbienen bilden Staaten und besitzen erstaunliche Fähigkeiten im Bereich der Luftaufklärung und Orientierung.

$4 \cdot 10^6$ Neurone ergeben eine *Maus*, hier sind wir schon in der Welt der Wirbeltiere angelangt.

$1.5 \cdot 10^7$ Neurone reichen bereits für eine *Ratte*, ein Tier, was als außerordentlich klug verschrien ist und oft als Teilnehmer für verschiedene anschauliche Intelligenztests bezüglich der Tierwelt herangezogen wird. Ratten können außerordentlich gut riechen und sich orientieren, zeigen außerdem Sozialverhalten. In etwa derselben Größenordnung liegt das Gehirn eines *Frosches*. Der Frosch besitzt einen komplizierten und funktionsreichen Körperbau, kann schwimmen und ist zu komplexen Verhaltensweisen fähig. Er kann besagte Fliege während einer Sprungbewegung im dreidimensionalen Raum durch seine Augen kontinuierlich zielerfasssen und durch seine Zunge mit vertretbarer Wahrscheinlichkeit fangen.

$5 \cdot 10^7$ Neurone ergeben eine *Fledermaus*. Die Fledermaus kann sich in totaler Dunkelheit rein akustisch in einem Raum orientieren, auf mehrere Zentimeter genau. Die Fledermaus kann sich selbst tarnende Insekten (z.B Motten tarnen sich durch eine bestimmte Flügelstruktur, welche Schall schlecht zurückwirft) während des Fluges akustisch orten und ebenso während des Fluges fressen.

$1.6 \cdot 10^8$ Neurone benötigt das Gehirn eines *Hundes*, der seit jeher Weggefährte des Menschen ist. Kommen wir zu einem weiteren beliebten Gefährten des Menschen:

$3 \cdot 10^8$ Neurone, also ca. doppelt so viele wie ein Hund, besitzt eine *Katze*. Wie wir wissen, sind Katzen sehr elegante, geduldige Raubtiere, fähig zu einer Vielzahl von Verhaltensweisen. In derselben Größenordnung liegt übrigens der *Octopus*, von dem nur wenige wissen, dass er z. B. in Labyrinth-Orientierungstests der Ratte deutlich überlegen ist.

Für $6 \cdot 10^9$ Neurone gibt es bereits einen Schimpansen, eines der Tiere, die dem Menschen schon sehr ähneln.

10^{11} Neurone besitzt ein *Mensch*. Der Mensch besitzt meist sehr umfangreiche kognitive Fähigkeiten und ist in der Lage zu sprechen, zu abstrahieren, zu erinnern und Werkzeuge sowie das Wissen von anderen Menschen zu nutzen, um fortgeschrittene Technologien und vielfältige soziale Strukturen zu entwickeln.

Mit $2 \cdot 10^{11}$ Neuronen gibt es Nervensysteme, die mehr Neuronen besitzen als die des Menschen: Hier seien Elephanten und bestimmte Walarten genannt.

Bereits mit der oben (sehr dünn) beschriebenen Rechenleistung einer Fliege können unsere aktuellen Computer schon nicht mehr mithalten. Neuere Forschungsergebnisse legen sogar nahe, dass die in Nervensystemen ablaufenden Prozesse noch einmal deutlich mächtiger sind, als man bis vor kurzem noch dachte: Michaeva et al. beschreiben eine eigene Synapsen-integrierte Informationsverarbeitung [MBW$^+$10]. Inwiefern sich das bestätigt, wird die Zeit zeigen.

2.5 Übergang zu technischen Neuronen: Neuronale Netze sind eine Karikatur der Biologie

Wie kommen wir nun von den biologischen Neuronalen Netzen zu den technischen? Durch radikalste Vereinfachung. Hierzu möchte ich nun die für die technische Seite relevanten Erkenntnisse noch einmal kurz zusammenfassen:

Wir haben gesehen, dass die biologischen Neurone miteinander auf gewichtete Weise vernetzt sind und ihr Signal bei Reizung elektrisch über das Axon übertragen. Von dort gelangen sie aber nicht direkt in die Nachfolgeneurone, sondern überwinden erst den synaptischen Spalt, in dem das Signal durch variierbare chemische Vorgänge noch einmal verändert wird. Im empfangenden Neuron werden dann die vielen durch den synaptischen Spalt nachverarbeiteten Inputs zu einem Impuls zusammengefasst bzw. aufkumuliert. Je nachdem, wie sehr das Neuron durch den kumulierten Input gereizt wird, gibt es dann selbst einen Impuls ab, oder nicht – der Output ist also *nichtlinear*, er ist nicht etwa proportional zum kumulierten Input. Unsere kurze Zusammenfassung entspricht nun genau den wenigen Faktoren biologischer Neuronaler Netze, welche wir in die technische Approximation übernehmen wollen:

Vektorieller Input: Der Input von technischen Neuronen hat viele Komponenten, ist also ein *Vektor*. In der Natur empfängt ein Neuron ja auch Impulse von durchschnittlich 10^3 bis 10^4 anderen Neuronen.

Skalarer Output: Der Output eines Neurons ist ein Skalar, hat also nur eine Komponente. Viele skalare Outputs bilden dann wieder einen vektoriellen Input eines

anderen Neurons. Dies bedeutet insbesondere, dass irgendwo im Neuron die vielen Inputkomponenten so zusammengefasst werden müssen, dass nur eine Komponente übrig bleibt.

Synapsen verändern Input: Auch in technischen Neuronalen Netzen werden die Inputs vorverarbeitet, nämlich mit einer Zahl (dem Gewicht) multipliziert – also *gewichtet*. Die Menge aller dieser synaptischen Gewichte stellen – sowohl im biologischen als auch im technischen Sinne – den Informationsspeicher des neuronalen Netzes dar.

Aufkumulieren der Inputs: In der Biologie werden die Inputs nach der chemischen Veränderung zu einem Impuls zusammengefasst, also aufkumuliert – in der Technik geschieht dies oft durch die gewichtete Summe, die wir noch kennenlernen werden. Im Neuron arbeiten wir also anstatt mit einem Vektor nach der Aufkumulierung mit nur *einem* Wert, einem Skalar, weiter.

Nichtlineare Kennlinie: Auch unsere technischen Neurone haben keinen zum Input proportionalen Output.

Gewichte einstellbar: Die Gewichte, mit denen die Inputs gewichtet werden, sind variabel, ähnlich den chemischen Prozessen am synaptischen Spalt. Dies verleiht dem Netz eine große Dynamik, denn in den Gewichten bzw. der Art und Stärke der chemischen Vorgänge in einem synaptischen Spalt wird das „Wissen“ eines Neuronalen Netzes zu großen Teilen gespeichert.

Unser aktuelles, nur salopp formuliertes, sehr einfaches Neuronenmodell erhält also einen vektoriellen Input

$$\vec{x},$$

mit Komponenten x_i. Es kumuliert diese multipliziert mit den zugehörigen Gewichten w_i auf:

$$\sum_i w_i x_i.$$

Obiger Term wird auch *gewichtete Summe* genannt. Die nichtlineare Abbildung f bestimmt dann den skalaren Output y:

$$y = f\left(\sum_i w_i x_i\right).$$

Nach dieser Überleitung wollen wir nun unser Neuronenmodell genauer spezifizieren und ihm noch einige Kleinigkeiten hinzufügen. Auf welche Weise Gewichte eingestellt werden können, werden wir anschließend ebenfalls betrachten.

Übungsaufgaben

Aufgabe 4. Es wird geschätzt, dass ein menschliches Gehirn etwa 10^{11} Nervenzellen besitzt, von denen jede etwa $10^3 - 10^4$ Synapsen aufweist. Gehen wir für diese Aufgabe von 10^3 Synapsen pro Neuron aus. Nehmen wir weiter an, eine einzelne Synapse könnte 4 Bit an Information speichern. Naiv gerechnet: Welche Speicherkapazität hat das Gehirn also? Beachten Sie, dass die Information, welches Neuron mit welchem anderen eine Verbindung besitzt, auch von Bedeutung ist.

Kapitel 3

Bausteine künstlicher Neuronaler Netze

Formale Definitionen und umgangssprachliche Erklärungen der Bestandteile, die die technischen Adaptionen der biologischen Neuronalen Netze ausmachen. Erste Beschreibungen, wie man diese Bestandteile zu einem Neuronalen Netz zusammensetzen kann.

Dieses Kapitel beinhaltet die formalen Definitionen für den Großteil der Bausteine Neuronaler Netze, die wir später verwenden werden. Nach dem Ende dieses Kapitels kann man ohne weiteres auch einzelne Kapitel des Skriptums lesen, ohne die vorangehenden zu betrachten (auch wenn das natürlich nützlich wäre).

3.1 Der Zeitbegriff bei Neuronalen Netzen

Bei manchen Definitionen verwenden wir in dieser Arbeit einen Begriff der *Zeit* bzw. der Durchlaufanzahl des Neuronalen Netzes. Die Zeitrechnung unterteilt sich hierbei in diskrete Zeitschritte:

Definition 3.1 (Zeitbegriff)**.** Die aktuelle Zeit (Jetzt-Zeit) wird dabei als (t) bezeichnet, der nächste ***Zeitschritt*** mit $(t+1)$, der vorherige mit $(t-1)$ und sämtliche anderen analog. Beziehen sich verschiedene mathematische Größen (z.B. net_j oder o_i) in den folgenden Kapiteln auf einen bestimmten Zeitpunkt, so lautet die Schreibweise hierfür z.B. $\text{net}_j(t-1)$ oder $o_i(t)$.

Biologisch ist das natürlich nicht sehr plausibel (bei unserem Gehirn wartet auch kein Neuron auf ein anderes), aber die Implementierung wird dadurch ganz wesentlich vereinfacht.

3.2 Bestandteile Neuronaler Netze

Ein technisches Neuronales Netz besteht aus simplen Recheneinheiten, den ***Neuronen***, sowie gerichteten, gewichteten Verbindungen zwischen diesen. Die Verbindungsstärke (bzw. das Verbindungs*gewicht*) zwischen zwei Neuronen i und j wollen wir als $w_{i,j}$ bezeichnen[1].

Definition 3.2 (Neuronales Netz)**.** Ein ***Neuronales Netz*** ist ein sortiertes Tripel (N, V, w) mit zwei Mengen N, V sowie einer Funktion w, wobei N die Menge der *Neurone* bezeichnet und V eine Menge $\{(i, j)|i, j \in \mathbb{N}\}$ ist, deren Elemente ***Verbindungen*** von Neuron i zu Neuron j heißen. Die Funktion $w : V \to \mathbb{R}$ definiert die ***Gewichte***, wobei $w((i, j))$, das Gewicht der Verbindung von Neuron i zu Neuron j, kurz mit $w_{i,j}$ bezeichnet wird. Sie ist je nach Auffassung entweder undefiniert oder 0 für Verbindungen, welche in dem Netz nicht existieren.

SNIPE: In Snipe instantiiert man zunächst ein Objekt der Klasse `NeuralNetworkDescriptor`, dass den groben Umriss eines Netzes definiert (z.B. die Anzahl der Neuronenschichten). Mittels des Descriptors kann man dann beliebig viele konkrete Neuronale Netze in Form von Objekten der Klasse `NeuralNetwork` instantiieren. Um mit der Snipe-Programmierung anzufangen, sind die Dokumentationen genau dieser beiden Klassen – in dieser Ordnung – der richtige Ort zum Lesen. Dieses Layout aus Descriptor und abhängigen konkreten Netzen ist implementierungstechnisch durchaus sinnvoll, denn man kann damit effizient auch große Mengen ähnlicher (aber nicht unbedingt gleicher) Netze erzeugen, und verschiedene Parameter zentral ändern.

Die Gewichtungen lassen sich also in einer quadratischen ***Gewichtsmatrix*** W oder wahlweise einem ***Gewichtsvektor*** W implementieren, wobei im Falle der Matrix die Zeilennummer angibt, *von wo* die Verbindung ausgeht, und die Spaltennummer, welches Neuron ihr *Ziel* ist. In diesem Fall markiert die Zahl 0 in der Tat eine nicht existierende Verbindung. Diese Matrixdarstellung wird auch ***Hinton-Darstellung*** genannt[2].

Die Neurone und Verbindungen sind ihrerseits aus folgenden Bestandteilen und Größen zusammengesetzt (ich gehe dabei den Weg, den die Daten innerhalb eines Neurons nehmen, der Abb. 3.1 auf der rechten Seite nach von oben nach unten):

1 Vorsicht beim Literaturstudium: Bei manchen Literaturstellen können die i und j in $w_{i,j}$ vertauscht sein – hier gibt es keinen einheitlichen Standard. Generell bemühe ich mich aber, in diesem Skriptum die Schreibweise zu finden, die ich häufiger und an prominenteren Literaturstellen antreffe.

2 Wieder Vorsicht bei Literaturstudium: Bei manchen Literaturstellen können Achsen und Zeilen vertauscht sein. Auch hier gibt es keine Konsistenz über die Gesamtliteratur.

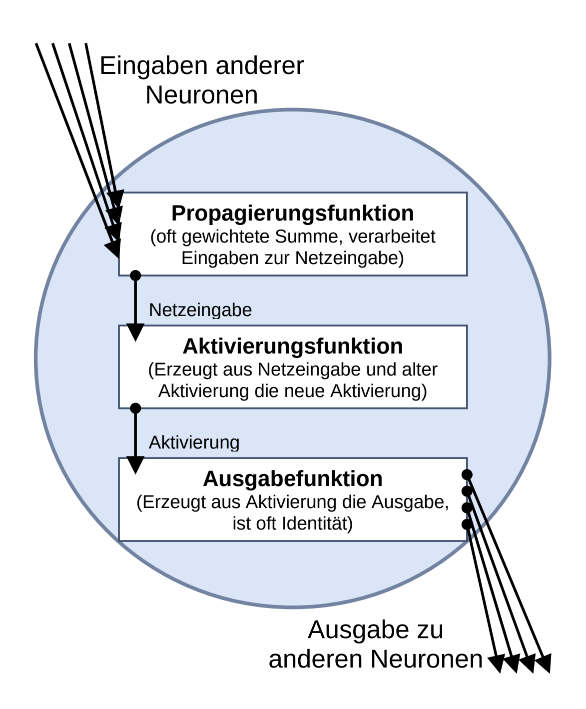


Abbildung 3.1: Datenverarbeitung eines Neurons. Die Aktivierungsfunktion eines Neurons beinhaltet den Schwellenwert.

3.2.1 Verbindungen übertragen Informationen, die von den Neuronen verarbeitet werden

Über die Verbindungen werden Daten zwischen Neuronen übertragen, wobei das Verbindungsgewicht entweder verstärkend oder hemmend wirkt. Die Definition von Verbindungen ist bereits in der Definition des Neuronalen Netzes eingeschlossen.

SNIPE: Verbindungsgewichte lassen sich mit der Methode `NeuralNetwork.setSynapse` ändern.

3.2.2 Die Propagierungsfunktion verwandelt vektorielle Eingaben zur skalaren Netzeingabe

Wenn man ein Neuron j betrachtet, so findet man meistens eine ganze Menge Neurone, von denen eine Verbindung zu j ausgeht, die also Ausgaben an j weiterleiten.

Die ***Propagierungsfunktion*** nimmt für ein Neuron j Ausgaben $o_{i_1}, \ldots, o_{i_n}$ anderer Neurone $i_1, i_2, \ldots, i_n$ entgegen (von denen eine Verbindung zu j existiert), und verarbeitet diese unter Berücksichtigung der Verbindungsgewichte $w_{i,j}$ zur *Netzeingabe* net_j, welche von der *Aktivierungsfunktion* weiterverwendet werden kann. Die ***Netzeingabe*** ist also das Ergebnis der Propagierungsfunktion.

Definition 3.3 (Propagierungsfunktion und Netzeingabe)**.** Es sei $I = \{i_1, i_2, \ldots, i_n\}$ die Menge der Neurone, bei denen gilt $\forall z \in \{1, \ldots, n\} : \exists w_{i_z,j}$. Dann berechnet sich die Netzeingabe von j, genannt net_j, durch die Propagierungsfunktion f_{prop}:

$$net_j = f_{\text{prop}}(o_{i_1}, \ldots, o_{i_n}, w_{i_1,j}, \ldots, w_{i_n,j}) \tag{3.1}$$

Beliebt ist hier die ***Gewichtete Summe***: Die Multiplikation der Ausgabe eines jeden i mit $w_{i,j}$, und die Aufsummierung der Ergebnisse:

$$\text{net}_j = \sum_{i \in I} (o_i \cdot w_{i,j}) \tag{3.2}$$

SNIPE: Die Gewichtete Summe ist exakt so in Snipe als Propagierungsfunktion realisiert.

3.2.3 Die Aktivierung ist der „Schaltzustand“ eines Neurons

Nach dem Vorbild der Natur ist jedes Neuron zu jeder Zeit zu einem bestimmten Grad *aktiv*, *gereizt*, oder was man sonst für Ausdrücke verwenden möchte. Von diesem *Aktivierungszustand* hängen die Reaktionen des Neurons auf Eingaben ab. Der Aktivierungszustand gibt also den Grad der Aktivierung eines Neurons an und wird oft kurz als ***Aktivierung*** bezeichnet. Seine formale Definition wird von der gleich folgenden Definition der *Aktivierungsfunktion* eingeschlossen. Allgemein kann man aber definieren:

Definition 3.4 (Aktivierungszustand / Aktivierung allgemein)**.** Sei j Neuron. Der Aktivierungszustand a_j, kurz Aktivierung genannt, ist j eindeutig zugeordnet, bezeichnet den Grad der Aktivität des Neurons und ergibt sich aus der Aktivierungsfunktion.

SNIPE: Der Aktivierungszustand von Neuronen lässt sich mit den Methoden `getActivation` bzw. `setActivation` der Klasse `NeuralNetwork` abrufen bzw. setzen.

3.2.4 Neuronen werden aktiviert, wenn die Netzeingabe ihren Schwellenwert überschreitet

Um den Schwellenwert herum reagiert die Aktivierungsfunktion eines Neurons besonders empfindlich. Biologisch gesehen stellt der Schwellenwert die Reizschwelle dar, ab der ein Neuron feuert. Der Schwellenwert wird ebenfalls weitestgehend in der Definition der Aktivierungsfunktion mitdefiniert, allerdings kann man allgemein definieren:

Definition 3.5 (Schwellenwert allgemein)**.** Sei j Neuron. Der ***Schwellenwert*** Θ_j ist j eindeutig zugeordnet und markiert die Stelle der größten Steigung der Aktivierungsfunktion.

3.2.5 Die Aktivierungsfunktion berechnet abhängig von Schwellenwert und Netzeingabe, wie stark ein Neuron aktiviert ist

Wie wir schon gehört haben, hängt die Aktivierung a_j eines Neurons j zu einem bestimmten Zeitpunkt davon ab, wie aktiviert das Neuron *bereits war*[3] und welche *Eingaben* es von außen erhalten hat.

3 Die vorherige Aktivierung muss nicht immer in die Berechnung des neuen Aktivierungszustandes miteinbezogen werden – wir werden für beide Varianten Beispiele kennenlernen.

Definition 3.6 (Aktivierungsfunktion und Aktivierung)**.** Sei j Neuron. Die ***Aktivierungsfunktion*** ist definiert als

$$a_j(t) = f_{\text{act}}(\text{net}_j(t), a_j(t-1), \Theta_j) \tag{3.3}$$

und verarbeitet also die *Netzeingabe* net_j und den *alten Aktivierungszustand* $a_j(t-1)$ zu einem *neuen Aktivierungszustand* $a_j(t)$, wobei der *Schwellenwert* Θ wie oben schon erläutert eine große Rolle spielt.

Im Unterschied zu anderen Größen innerhalb des Neuronalen Netzes (insbesondere zu den bisher definierten) ist die Aktivierungsfunktion oft *global für alle oder zumindest eine Menge von Neuronen* definiert, nur die Schwellenwerte unterscheiden sich dann von Neuron zu Neuron. Auch sollten wir im Hinterkopf behalten, dass sich die Schwellenwerte z.B. durch einen Lernvorgang ändern sollten, so dass es insbesondere nötig werden kann, die Schwellenwerte auf die Zeit zu beziehen und z.B. Θ_j als $\Theta_j(t)$ zu schreiben (das habe ich hier der Übersichtlichkeit zuliebe erst einmal unterlassen). Die Aktivierungsfunktion wird auch oft als ***Transferfunktion*** bezeichnet.

SNIPE: Aktivierungsfunktionen sind in Snipe generalisiert zu „Neuronenverhaltensweisen“ (*Neuron Behaviors*). Diese können zum einen ganz normale Aktivierungsfunktionen repräsentieren, aber auch interne Zustände besitzen. Diesbezügliche Programmbestandteile befinden sich im Paket `neuronbehavior`, wo auch einige der gleich vorgestellten Aktivierungsfunktionen implementiert sind. Das Interface `NeuronBehavior` erlaubt die Implementierung eigener Verhaltensweisen. Objekte, die von diesem Interface erben, können einem `NeuralNetworkDescriptor` übergeben werden; pro Neuronenschicht kann eine Neuronenverhaltensweise festgelegt werden.

3.2.6 Gängige Aktivierungsfunktionen

Die einfachste Aktivierungsfunktion ist die ***binäre Schwellenwertfunktion*** (Abb. 3.2 auf Seite 48), welche nur zwei Werte annehmen kann (auch ***Heaviside-Funktion*** genannt). Sie wechselt am Schwellenwert von einem Wert auf den andern, ist aber ansonsten konstant. Dies impliziert, dass sie am Schwellenwert nicht differenzierbar ist und die Ableitung ansonsten 0 ist. Dies macht z.B. das Lernen mit Backpropagation unmöglich (später mehr dazu). Beliebt ist weiterhin die auch ***Logistische Funktion*** genannte ***Fermifunktion*** (Abb. 3.2)

$$\frac{1}{1+\mathrm{e}^{-x}}, \tag{3.4}$$

welche einen Wertebereich von $(0,1)$ aufweist, sowie der ***Tangens Hyperbolicus*** (Abb. 3.2) mit einem Wertebereich von $(-1,1)$ – beide differenzierbar. Die Fermifunktion ist einfach um einen ***Temperaturparameter*** T zu der Form

$$\frac{1}{1+e^{\frac{-x}{T}}} \tag{3.5}$$

erweiterbar, der, je kleiner man ihn wählt, die Funktion auf der x-Achse zusammenstaucht. So kann man sie beliebig an die Heaviside-Funktion annähern. Es existieren übrigens auch Aktivierungsfunktionen, welche nicht eindeutig bestimmt sind, sondern nach einer Zufallsverteilung von der Eingabe abhängen (*stochastische Aktivierungsfunktionen*).

Eine wirklich erwähnenswerte Alternative zum Tangens Hyperbolicus wurde von Anguita et al. vorgeschlagen [APZ93]. Inspiriert von den eher langsamen Computern, die es 1993 gab, haben sie sich Gedanken gemacht, wie man ein Neuronales Netz beschleunigen könnte, und kamen schnell darauf, dass die Approximation der e-Funktion im Tangens Hyperbolicus rechenzeitmäßig sehr lange dauert. So haben sie den Tangens Hyperbolicus näherungsweise mit zwei Parabelbögen und zwei Halbgeraden nachgebaut. Die resultierende Funktion bietet zwar statt des Werteberichs von $[-1;1]$ „nur“ einen Wertebereich von $[-0.96016; 0.96016]$, lässt sich aber – je nach CPU – um den Faktor 200 schneller mittels einer Addition und zwei Multiplikationen berechnen und bietet auch noch weitere Vorteile, die an anderer Stelle genannt werden.

SNIPE: Die hier vorgestellten stetigen Aktivierungsfunktionen finden sich in den Klassen `Fermi` sowie `TangensHyperbolicus` im Paket `neuronbehavior` wieder. Die schnelle Approximation des Tangens Hyperbolicus befindet sich in der Klasse `TangensHyperbolicusAnguita`.

3.2.7 Eine Ausgabefunktion kann genutzt werden, um die Aktivierung nochmals zu verarbeiten

Die ***Ausgabefunktion*** eines Neurons j berechnet die Werte, die an die anderen Neurone, zu welchen eine Verbindung von j besteht, weitergegeben werden. Formaler:

Definition 3.7 (Ausgabefunktion)**.** Sei j Neuron. Die Ausgabefunktion

$$f_{\text{out}}(a_j) = o_j \tag{3.6}$$

berechnet dann den Ausgabewert o_j des Neurons j aus seinem *Aktivierungszustand* a_j.

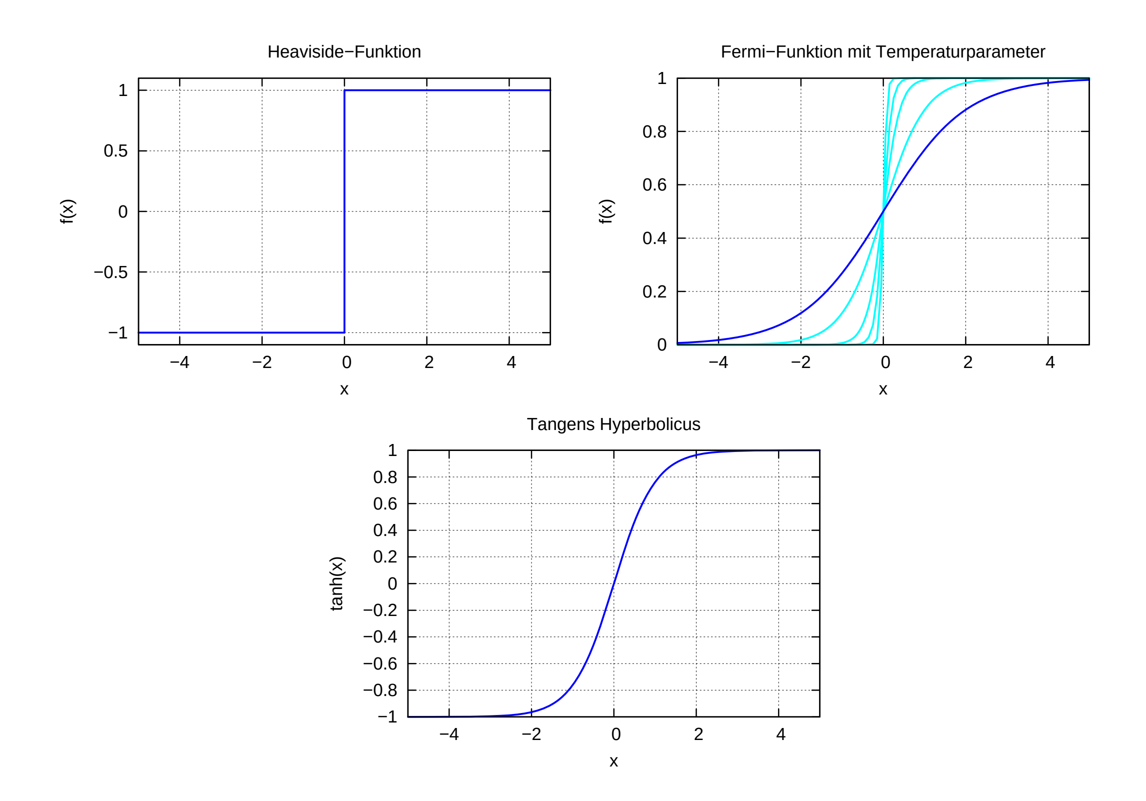


Abbildung 3.2: Verschiedene gängige Aktivierungsfunktionen, von oben nach unten: Heaviside- bzw. Binäre Schwellenwertfunktion, Fermifunktion, Tangens hyperbolicus. Die Fermifunktion wurde um einen Temperaturparameter erweitert. Die ursprüngliche Fermifunktion ist hierbei dunkel herausgestellt, die Temperaturparameter bei den modifizierten Fermifunktionen betragen (aufsteigend geordnet nach Anstieg) $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$ und $\frac{1}{25}$.

Auch die Ausgabefunktion ist i.d.R. global definiert. Oft ist diese Funktion die *Identität*, d.h. es wird direkt die *Aktivierung* a_j ausgegeben[4]:

$$f_{\text{out}}(a_j) = a_j, \text{ also } o_j = a_j \tag{3.7}$$

Solange nicht explizit anders angegeben, werden wir innerhalb dieser Arbeit die Identität als Ausgabefunktion verwenden.

3.2.8 Lernverfahren passen ein Netz auf unsere Bedürfnisse an

Da wir uns diesem Thema später noch sehr ausführlich widmen werden und erst einmal die Grundsätze des Aufbaus Neuronaler Netze kennen lernen möchten, sei hier nur kurz und allgemein definiert:

Definition 3.8 (Lernregel, allgemein)**.** Ein ***Lernverfahren*** ist ein Algorithmus, der das Neuronale Netz verändert und ihm so beibringt, für eine vorgegebene Eingabe eine gewünschte Ausgabe zu produzieren.

3.3 Verschiedene Netztopologien

Nachdem wir nun den Aufbau der Bestandteile Neuronaler Netze kennengelernt haben, möchte ich einen Überblick über die gängigen Topologien (= Bauarten) von Neuronalen Netzen geben – also aus den Bausteinen Netze bauen. Jede beschriebene Topologie wird sowohl mit Abbildung als auch durch seine Hinton-Darstellung illustriert, damit der Leser sofort die Charakteristika sehen und bei der Betrachtung anderer Netze anwenden kann.

Die gepunktet eingezeichneten Gewichte werden in der Hinton-Darstellung als hellgraue Kästchen dargestellt, die durchgezogenen als dunkelgraue. Die der Übersichtlichkeit halber hinzugefügten Input- und Outputpfeile sind in der Hinton-Darstellung nicht zu finden. Um zu verdeutlichen, dass die Verbindungen von den Zeilenneuronen zu den Spaltenneuronen gehen, ist in der oberen linken Zelle der kleine Pfeil ↱ eingefügt.

SNIPE: Snipe ist dafür ausgelegt, verschiedenste Netztopologien realisieren zu können. Hierfür unterscheidet Snipe verschiedene Synapsenklassen (abhängig von Synapsenstart und -Ziel), die in einer `NeuralNetworkDescriptor`-Instanz für die davon abhängenden Netze nach Belieben mittels der `setAllowed`-Methoden erlaubt oder verboten werden können.

4 Andere Definitionen der Ausgabe können sinnreich sein, wenn der Wertebereich einer Aktivierungsfunktion nicht ausreichend ist.

3.3.1 FeedForward-Netze bestehen aus Schichten und Verbindungen zur jeweils nächsten Schicht

FeedForward-Netze (Abb. 3.3 auf der rechten Seite) sind in dieser Arbeit die Netze, die wir zunächst erforschen werden (wenn auch später andere Topologien verwendet werden). Die Neurone sind in ***Schichten*** eingeteilt: Eine ***Eingabeschicht***, n ***versteckte Verarbeitungsschichten*** (unsichtbar von außen, weswegen man die Neurone darin auch als *versteckt* bezeichnet) und eine ***Ausgabeschicht***. Die Verbindungen von einem jeden Neuron dürfen bei einem FeedForward-Netz ausschließlich ein Neuron der nächsten Schicht (in Richtung Ausgabeschicht) treffen. Die für ein FeedForward-Netz zugelassenen Verbindungen sind in Abb. 3.3 auf der rechten Seite durchgezogen dargestellt. Sehr oft begegnet man FeedForward-Netzen, in denen jedes Neuron i eine Verbindung zu allen Neuronen der nachfolgenden Schicht besitzt (diese Schichten nennt man dann untereinander ***vollverknüpft***). Outputneurone werden zur Vermeidung von Benennungskonflikten oft mit Ω bezeichnet.

Definition 3.9 (FeedForward-Netz)**.** Ein FeedForward-Netz (Abb. 3.3 auf der rechten Seite) besitzt klar abgetrennte Schichten von Neuronen: Eine Eingabeschicht, eine Ausgabeschicht und beliebig viele innere, von außen nicht sichtbare Verarbeitungsschichten (auch versteckte Schichten bzw. hidden layer genannt). Verbindungen sind nur zu Neuronen der jeweils nächsten Schicht erlaubt.

3.3.1.1 ShortCut-Verbindungen überspringen Schichten

Manche FeedForward-Netze gestatten sog. ***ShortCut-Connections*** (Abb. 3.4 auf Seite 52): Verbindungen, welche eine oder mehrere Ebenen überspringen. Auch diese Verbindungen dürfen ausschließlich in Richtung der Ausgabeschicht zeigen.

Definition 3.10 (FeedForward-Netz mit ShortCut-Connections)**.** Wie beim FeedForward-Netz, doch dürfen Verbindungen nicht nur die nächste, sondern auch jede andere nachfolgende Schicht zum Ziel haben.

3.3.2 Rückgekoppelte Netze beeinflussen sich selbst

Man spricht von einer ***Rückkopplung*** oder ***Rekurrenz***, wenn ein Neuron sich auf irgendeine Weise oder durch irgendeinen Verbindungsweg selbst beeinflussen kann. Bei rückgekoppelten oder rekurrenten Netzen sind nicht immer Input- oder Outputneurone explizit definiert, daher lasse ich die diesbezügliche Beschriftung in den Abbildungen weg und nummeriere die Neurone nur durch.

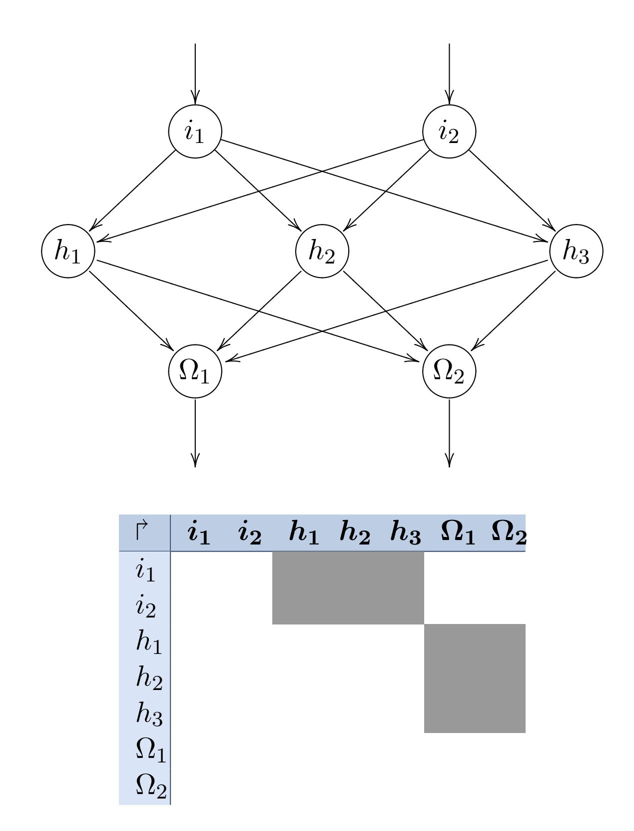


Abbildung 3.3: Ein FeedForward-Netz mit drei Schichten: Zwei Inputneurone, drei versteckte Neurone und zwei Outputneurone. Charakteristisch in der Hinton-Darstellung für vollverknüpfte FeedForward-Netze: Die Blockbildung über der Diagonalen.

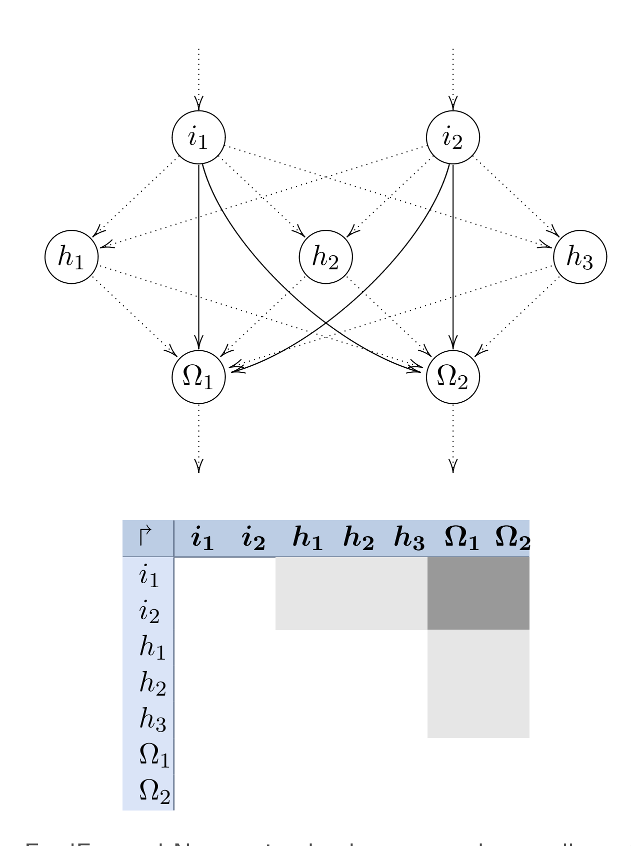


Abbildung 3.4: Ein FeedForward-Netz mit durchgezogen dargestellten Shortcut-Connections. Rechts der FeedForward-Blöcke sind in der Hinton-Darstellung neue Verbindungen hinzugekommen.

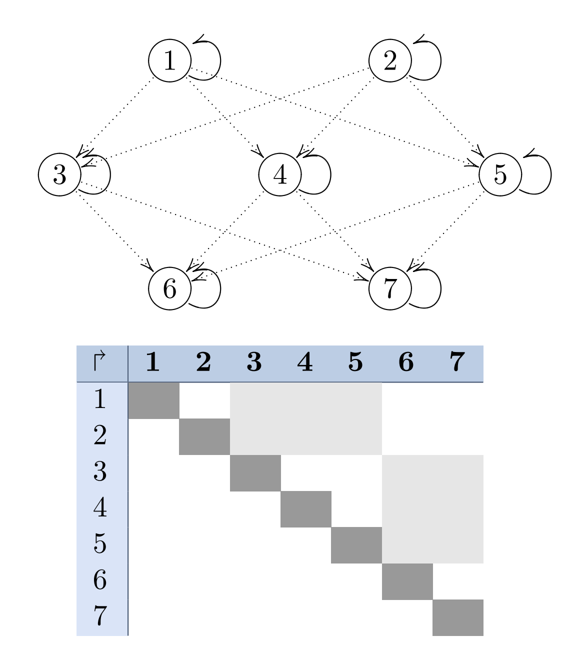


Abbildung 3.5: Ein FeedForward-ähnliches Netz mit direkt rückgekoppelten Neuronen. Die direkten Rückkopplungen sind durchgezogen dargestellt. Sie entsprechen in der Hinton-Darstellung genau der Diagonalen der Matrix.

3.3.2.1 Direkte Rückkopplungen starten und enden an demselben Neuron

Manche Netze lassen Verbindungen von einem Neuron zu sich selbst zu, was als ***direkte Rückkopplung*** (manchmal auch *Selbstrückkopplung* oder *Selbstrekurrenz*) bezeichnet wird (Abb. 3.5). Neurone hemmen und stärken sich so selbst, um an ihre Aktivierungsgrenzen zu gelangen.

Definition 3.11 (Direkte Rückkopplung)**.** Wir erweitern wieder das FeedForward-Netz, diesmal um Verbindungen von einem Neuron j zu *sich selbst*, deren Gewichte dann den Namen $w_{j,j}$ tragen. Anders ausgedrückt darf die Diagonale der Gewichtsmatrix W ungleich 0 sein.

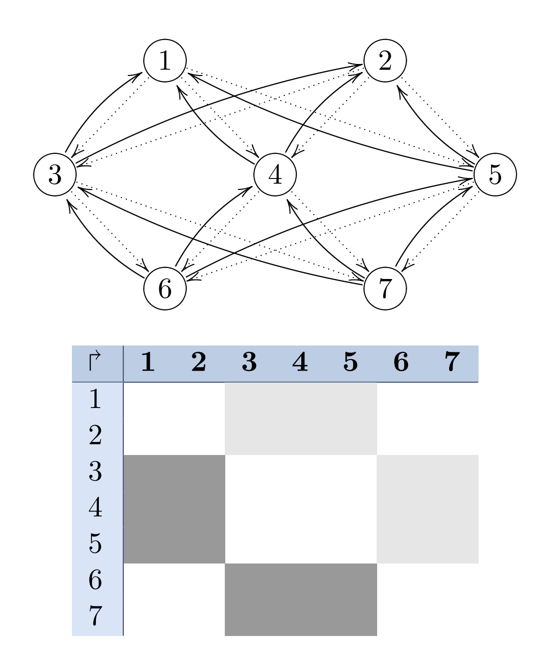


Abbildung 3.6: Ein FeedForward-ähnliches Netz mit indirekt rückgekoppelten Neuronen. Die indirekten Rückkopplungen sind durchgezogen dargestellt. Wie wir sehen, können hier auch Verbindungen zu vorherigen Schichten existieren. Die zu den FeedForward-Blöcken symmetrischen Felder in der Hinton-Darstellung sind nun belegt.

3.3.2.2 Indirekte Rückkopplungen beeinflussen ihr Startneuron nur über Umwege

Sind Verbindungen in Richtung der Eingabeschicht gestattet, so nennt man diese ***indirekte Rückkopplungen***. Ein Neuron j kann sich dann durch Umwege nach vorne selbst beeinflussen, indem es z.B. die Neurone der nächsten Schicht beeinflusst und die Neurone dieser nächsten Schicht wieder j (Abb. 3.6).

Definition 3.12 (Indirekte Rückkopplung)**.** Wieder vom FeedForward-Netz ausgehend, sind diesmal zusätzlich Verbindungen von Neuronen zur *vorherigen Schicht* erlaubt, also dürfen die Werte im Bereich unter der Diagonalen von W ungleich 0 sein.

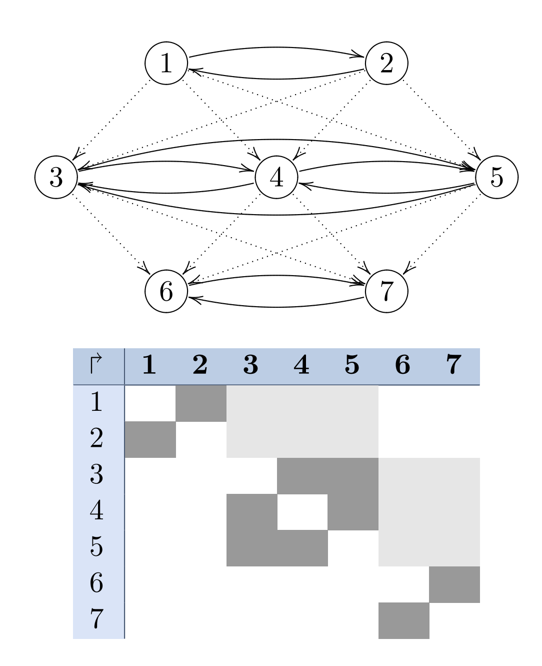


Abbildung 3.7: Ein FeedForward-ähnliches Netz mit lateral rückgekoppelten Neuronen. Die direkten Rückkopplungen sind durchgezogen dargestellt. Rückkopplungen existieren hier nur ebenenweise. In der Hinton-Darstellung sammeln sich um die Diagonale gefüllte Quadrate in Höhe der FeedForward-Blöcke an, die jedoch die Diagonale frei lassen.

3.3.2.3 Laterale Rückkopplungen verbinden Neuronen in ein- und derselben Ebene

Verbindungen von Neuronen *innerhalb einer Ebene* heißen ***laterale Rückkopplungen*** (Abb. 3.7). Oft hemmt hier jedes Neuron die anderen Neurone der Ebene und verstärkt sich selbst, es wird dann nur das stärkste Neuron aktiv (***Winner-Takes-All-Schema***).

Definition 3.13 (Laterale Rückkopplung)**.** Ein lateral rückgekoppeltes Netz erlaubt Verbindungen *innerhalb* einer Schicht.

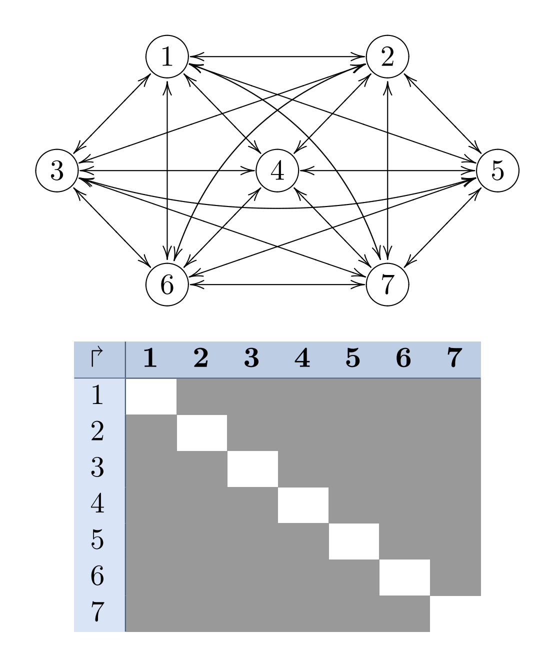


Abbildung 3.8: Ein vollständig verbundenes Netz mit symmetrischen Verbindungen, ohne direkte Rückkopplungen. Nur die Diagonale in der Hinton-Darstellung bleibt frei.

3.3.3 Vollständig verbundene Netze erlauben jede denkbare Verbindung

Vollständig verbundene Netze erlauben Verbindungen zwischen allen Neuronen, außer direkten Rückkopplungen; außerdem müssen die Verbindungen symmetrisch sein (Abb. 3.8). Ein populäres Beispiel sind die *selbstorganisierenden Karten*, welche in Kap. 10 vorgestellt werden.

Definition 3.14 (Vollständiger Verbund)**.** Hier darf prinzipiell jedes Neuron zu jedem eine Verbindung unterhalten – allerdings kann so auch jedes Neuron Eingabeneuron werden, weshalb direkte Rückkopplungen i.d.R. hier nicht angewandt werden, und es keine klar definierten Schichten mehr gibt. Die Matrix W darf also überall ungleich 0 sein, außer auf ihrer Diagonalen.

3.4 Das Biasneuron ist ein technischer Trick, Schwellwerte als Verbindungsgewichte zu behandeln

Wir wissen mittlerweile, dass Neurone in vielen Netzparadigmen einen *Schwellenwert* besitzen, der angibt, ab wann ein Neuron aktiv ist. Der Schwellenwert ist also ein Parameter der Aktivierungsfunktion eines Neurons. Dies ist zwar biologisch am plausibelsten, es ist jedoch kompliziert, zwecks Training des Schwellenwerts zur Laufzeit auf die Aktivierungsfunktion zuzugreifen.

Man kann allerdings Schwellenwerte $\Theta_{j_1}, \dots, \Theta_{j_n}$ für Neurone $j_1, j_2, \dots, j_n$ auch als *Gewicht einer Verbindung von einem immer feuernden Neuron* realisieren: Zu diesem Zweck integriert man ein zusätzliches, immer 1 ausgebendes Biasneuron in das Netz, verbindet es mit den Neuronen $j_1, j_2, \dots, j_n$ und gibt diesen neuen Verbindungen die Gewichte $-\Theta_{j_1}, \dots, -\Theta_{j_n}$, also die negativen Schwellenwerte.

Definition 3.15. Ein ***Biasneuron*** ist ein Neuron, welches immer 1 ausgibt und als



dargestellt wird. Es wird verwendet, um Schwellwerte als Gewichte zu repräsentieren, so dass beliebige Lernverfahren sie direkt mitsamt den Gewichten trainieren können.

Den Schwellenwert der Neurone $j_1, j_2, \dots, j_n$ setzt man dann auf 0. Nun sind die Schwellenwerte als Verbindungsgewichte implementiert (Abb. 3.9 auf der folgenden Seite) und können beim Training von Verbindungsgewichten direkt mittrainiert werden, was uns das Lernen erheblich vereinfacht.

Anders ausgedrückt: Wir verschieben die Einrechnung des Schwellenwerts von der Aktivierungsfunktion in die Propagierungsfunktion. Noch kürzer: Der Schwellenwert wird jetzt einfach von der Netzeingabe subtrahiert, ist also Teil der Netzeingabe. Formaler:

Seien $j_1, j_2, \dots, j_n$ Neurone mit Schwellenwerten $\Theta_{j_1}, \dots, \Theta_{j_n}$. Durch Einsetzen eines immer 1 ausgebenden Biasneurons, Erstellen von Verbindungen von diesem zu den Neuronen $j_1, j_2, \dots, j_n$ und Gewichtung dieser Verbindungen $w_{\texttt{BIAS},j_1}, \dots, w_{\texttt{BIAS},j_n}$ mit $-\Theta_{j_1}, \dots, -\Theta_{j_n}$ kann man $\Theta_{j_1} = \dots = \Theta_{j_n} = 0$ setzen und erhält ein äquivalentes Neuronales Netz, bei dem sämtliche Schwellenwerte durch Verbindungsgewichte realisiert sind.

Der Vorteil des Biasneurons ist ohne Zweifel die vielfach einfachere Implementierung des Netzes. Als Nachteil sei genannt, dass die Darstellung des Netzes schon bei nur

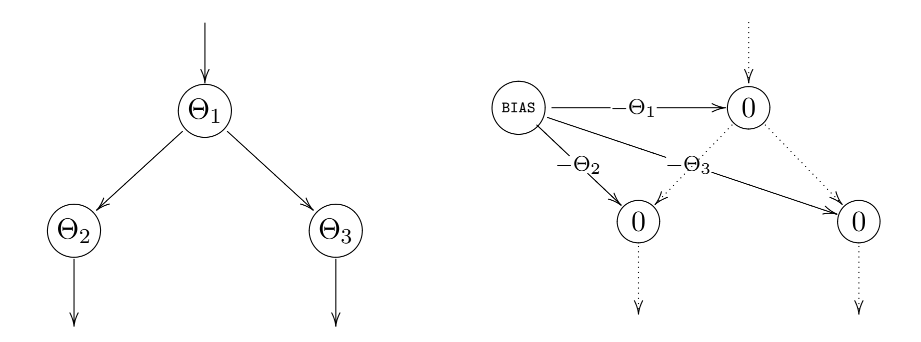


Abbildung 3.9: Zwei äquivalente Neuronale Netze, links eins ohne, rechts eins mit Biasneuron. Die Neuronenschwellenwerte stehen in den Neuronen, Verbindungsgewichte an den Verbindungen. Der Übersichtlichkeit zuliebe habe ich Gewichte der schon vorhandenen (rechts gepunktet dargestellten) Verbindungen nicht extra aufgeschrieben.

wenigen Neuronen recht unansehnlich wird, von einer großen Neuronenanzahl ganz zu schweigen. Übrigens wird ein Biasneuron auch oft ***On-Neuron*** genannt.

Wir wollen mit dem Biasneuron so verfahren, dass es ab hier der Übersichtlichkeit halber nicht mehr mit abgebildet wird, wir aber wissen, dass es so etwas gibt und dass man die Schwellenwerte damit einfach als Gewichte behandeln kann.

SNIPE: Auch in Snipe wurde anstatt Schwellwerten ein Biasneuron implementiert, es trägt den Index 0.

3.5 Darstellung von Neuronen

Wir haben oben bereits gesehen, dass man in Neuronen ihren Namen oder aber ihren Schwellenwert schreiben kann. Eine weitere sinnvolle Darstellungsmethode, welche wir auch in der weiteren Arbeit mehrfach anwenden werden, ist, Neurone nach ihrer Datenverarbeitungsart darzustellen. Einige Beispiele ohne weitere Erläuterungen seien in Abb. 3.10 auf der rechten Seite gegeben – erläutert werden die Neuronenarten, sobald sie gebraucht werden.

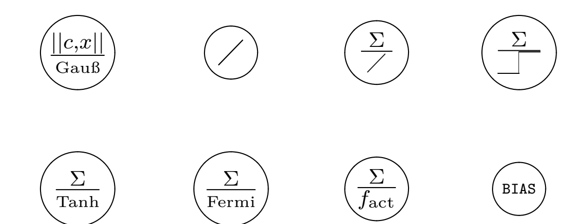


Abbildung 3.10: Verschiedene Neuronenarten, denen wir im Text noch begegnen werden.

3.6 Es ist nicht egal, in welcher Reihenfolge Neuronenaktivierungen berechnet werden

Für ein Neuronales Netz ist es von großer Bedeutung, in welcher ***Reihenfolge*** die einzelnen Neurone ihre Eingaben empfangen, verarbeiten, und Ergebnisse ausgeben. Hierbei werden zwei Modellklassen unterschieden:

3.6.1 Synchrone Aktivierung

Alle Neurone ändern ihre Werte ***synchron***, berechnen also simultan Netzeingaben, Aktivierung und Ausgabe und geben diese weiter. Synchrone Aktivierung kommt dem biologischen Vorbild am nächsten, macht aber – wenn sie in Hardware implementiert werden soll – nur auf bestimmten Parallelrechnern Sinn und speziell keinen Sinn für FeedForward-Netze. Diese Variante ist die allgemeinste und kann mit Netzen beliebiger Topologie sinnvoll angewandt werden.

Definition 3.16 (Synchrone Aktivierung)**.** Alle Neurone eines Netzes berechnen gleichzeitig Netzeingaben mittels Propagierungsfunktion, Aktivierung mittels Aktivierungsfunktion und Ausgabe mittels Ausgabefunktion. Hiernach ist der Aktivierungszyklus abgeschlossen.

SNIPE: Softwaretechnisch würde man diese sehr allgemeine Aktivierungsreihenfolge realisieren, indem man in jedem neuen Zeitschritt zunächst alle Netzeingaben aus den existierenden Aktivierungen berechnet, zwischenspeichert, und anschließend aus den Netzeingaben alle Aktivierungen. Und so ist das auch genau das, was in Snipe passiert, denn Snipe muss ja beliebige Topologien realisieren können.

3.6.2 Asynchrone Aktivierung

Hier ändern die Neurone ihre Werte nicht simultan, sondern zu verschiedenen Zeitpunkten. Hierfür gibt es verschiedene Ordnungen, von denen ich ausgewählte vorstelle:

3.6.2.1 Zufällige Ordnung

Definition 3.17 (Zufällige Aktivierungsordnung)**.** Bei der ***zufälligen Aktivierungsordnung*** wird jeweils ein Neuron i zufällig gewählt und dessen net_i, a_i und o_i aktualisiert. Bei n vielen Neuronen ist ein Zyklus die n-malige Durchführung dieses Schrittes. Offensichtlich werden manche Neurone pro Zyklus mehrfach aktualisiert, andere hingegen gar nicht.

Es ist klar, dass diese Aktivierungsordnung nicht immer sinnreich ist.

3.6.2.2 Zufällige Permutation

Bei der ***Zufälligen Permutation*** wird pro Zyklus jedes Neuron genau einmal berücksichtigt, das Ganze allerdings in zufälliger Reihenfolge.

Definition 3.18 (Zufällige Permutation)**.** Zunächst wird eine Permutation der Neurone zufällig berechnet, welche die Aktivierungsreihenfolge festlegt. In dieser Reihenfolge werden die Neurone dann sukzessive abgearbeitet.

Auch diese Aktivierungsreihenfolge wird nicht oft eingesetzt, weil erstens die Reihenfolge im Allgemeinen nicht sinnvoll ist und es zweitens sehr zeit- bzw. rechenaufwändig ist, bei jedem Zyklus eine neue Permutation zu berechnen. Wir werden in Form der *Hopfield-Netze* (Kap. 8) zwar eine Topologie kennenlernen, welche nominell eine zufällige oder zufällig permutierte Aktivierungsreihenfolge besitzt – die praktische Umsetzung sieht dort allerdings so aus, dass man aus o.g. Gründen eine feste Reihenfolge verwendet.

Bei allen Ordnungen können wahlweise entweder die alten Aktivierungen der Neurone des Zeitpunkts t als Ausgangspunkt genommen werden, oder aber, wo vorhanden, bereits die Aktivierungen des Zeitpunkts $t + 1$, für den wir eigentlich gerade die Aktivierungen errechnen.

3.6.2.3 Topologische Ordnung

Definition 3.19 (Topologische Aktivierung)**.** Bei der ***Topologischen Aktivierungsordnung*** werden die Neurone pro Zyklus in fester Ordnung aktualisiert, welche durch die *Netztopologie* definiert ist.

Dieses Verfahren kommt nur für *zyklenfreie*, also rückkopplungsfreie Netze in Frage, da man sonst keine Aktivierungsreihenfolge finden kann. In *FeedForward-Netzen* (für die das Verfahren äußerst günstig ist) würden so erst die Eingabeneurone aktualisiert, danach die inneren Neurone, als letztes die Ausgabeneurone. Dies spart eine Menge Zeit: Bei synchroner Aktivierung würde man für ein Feed-Forward-Netz mit drei Schichten beispielsweise drei vollständige Propagierungszyklen benötigen, damit eine Eingabe auch Auswirkungen auf die Ausgabeneurone haben kann. Mittels topologischer Aktivierung ist dies mit nur einem Propagierungszyklus geschafft. Allerdings lässt sich nicht für jede Netztopologie eine sinnvolle Ordnung finden, um auf diese Weise Zeit zu sparen.

SNIPE: Wenn der Leser mit Snipe FeedForward-Netze realisieren und auf diese Weise Rechenzeit einsparen möchte, so kann er in der Dokumentation zur Klasse `NeuralNetworkDescriptor` nach der Funktion ***Fastprop*** suchen und diese aktivieren. Während der Propagierung werden die Neurone dann nacheinander durchgegangen und für jedes Neuron Netzeingabe und Aktivierung auf einmal berechnet. Da die Neurone von Eingabeschicht über die verschiedenen Schichten bis hin zur Ausgabeschicht durchgehend aufsteigend nummeriert sind, entspricht diese Propagierungsvariante der topologisch sinnvollen Ordnung für FeedForward-Netze.

3.6.2.4 Feste Aktivierungsordnungen in der Implementierung

Offensichtlich kann man sich auch ***feste Aktivierungsordnungen*** definieren. Es ist daher eine beliebte Methode bei der Implementierung von z.B. FeedForward-Netzen, die Aktivierungsreihenfolge *einmal* nach der Topologie zu ermitteln und zur Laufzeit diese ermittelte Reihenfolge ohne weitere Prüfung weiter zu verwenden. Dies ist jedoch bei Netzen, die ihre Topologie verändern können, nicht unbedingt sinnvoll.

3.7 Kommunikation mit der Außenwelt: Ein- und Ausgabe von Daten in und von Neuronalen Netzen

Zuletzt sei noch betrachtet, dass in viele Arten von Neuronalen Netzen natürlich auch Daten eingegeben werden können. Diese werden dann verarbeitet und können eine

Ausgabe hervorrufen. Betrachten wir beispielsweise das FeedForward-Netz aus Abb. 3.3 auf Seite 51: Es hat zwei Eingabe- und zwei Ausgabeneurone, also auch zwei numerische Eingaben x_1, x_2 und Ausgaben y_1, y_2. Diese Schreibweise wird natürlich für Netze mit vielen Ein- und Ausgabeneuronen mühselig – insofern machen wir es uns einfach und fassen die Ein- und Ausgabekomponenten für n Ein- bzw. Ausgabeneurone in den Vektoren $x = (x_1, x_2, \ldots, x_n)$ und $y = (y_1, y_2, \ldots, y_n)$ zusammen.

Definition 3.20 (Eingabevektor)**.** Ein Netz mit n vielen Eingabeneuronen benötigt n Eingaben $x_1, x_2, \ldots, x_n$. Wir fassen diese als ***Eingabevektor*** $x = (x_1, x_2, \ldots, x_n)$ auf. Die ***Eingabedimension*** bezeichnen wir also mit n. Daten werden in ein Neuronales Netz eingegeben, indem die Komponenten des Eingabevektors einfach bei den Eingabeneuronen als Netzeingabe verwendet werden.

Definition 3.21 (Ausgabevektor)**.** Ein Netz mit m vielen Ausgabeneuronen liefert m Ausgaben $y_1, y_2, \ldots, y_m$. Wir fassen diese als ***Ausgabevektor*** $y = (y_1, y_2, \ldots, y_m)$ auf. Die ***Ausgabedimension*** bezeichnen wir also mit m. Daten werden von einem Neuronalen Netz ausgegeben, indem die Komponenten des Ausgabevektors von den Ausgabewerten der Ausgabeneurone übernommen werden.

SNIPE: Um Daten durch eine `NeuralNetwork`-Instanz zu propagieren, wird die `propagate`-Methode genutzt. Sie nimmt den Eingabevektor als Array von Doubles entgegen und liefert einen ebensolchen Ausgabevektor.

Wir haben nun die Grundbausteine der Neuronalen Netze definiert und näher betrachtet – ohne jedoch ein Netz einmal in Aktion zu sehen. Wir wollen mit dieser rein erklärenden Ansicht zunächst etwas weiter fortfahren und ganz allgemein beschreiben, wie ein Neuronales Netz lernen könnte.

Übungsaufgaben

Aufgabe 5. Ist es (Ihrer Meinung nach) sinnvoll, bei schichtenbasierten Netzen wie z.B. FeedForward-Netzen ein Biasneuron *pro Schicht* einzufügen? Erörtern Sie dies in Bezug auf die Darstellung und die Implementierung des Netzes. Ändert sich etwas am Ergebnis des Netzes?

Aufgabe 6. Zeigen Sie sowohl für die Fermifunktion $f(x)$, als auch für den Tangens Hyperbolicus $\tanh(x)$, dass sich die Ableitungen der Funktionen durch die Funktion selbst ausdrücken lassen, dass also die beiden Behauptungen

1. $f'(x) = f(x) \cdot (1 - f(x))$ und

2. $\tanh'(x) = 1 - \tanh^2(x)$

gelten.



Kapitel 4

Grundlagen zu Lernprozess und Trainingsbeispielen

Ansätze und Gedanken, auf welche Arten Maschinen etwas beizubringen ist. Korrigiert man Neuronale Netze? Bestärkt man sie nur? Lässt man sie gar ganz alleine ohne Hilfe lernen? Gedanken darüber, was wir während des Lernvorganges überhaupt verändern möchten, wie wir es verändern, über Fehlermessung und wann wir mit dem Lernen fertig sind.

Wie schon beschrieben, besteht das interessanteste Merkmal Neuronaler Netze in ihrer Fähigkeit, sich Problemen durch Training vertraut zu machen und, nach ausreichendem Training, auch bis dato unbekannte Probleme derselben Klasse lösen zu können, was man als ***Generalisierung*** bezeichnet. Bevor wir konkrete Lernverfahren kennenlernen, möchte ich in diesem Kapitel zunächst grundsätzliche Gedanken zum Lernprozess anregen.

4.1 Es gibt verschiedene Paradigmen zu lernen

Lernen ist ein weiter Begriff. Es bedeutet, das ein System sich in irgendeiner Form verändert, um sich z.B. an Veränderungen in seiner Umwelt anzupassen. Grundsätzlich verändert sich ein Neuronales Netz mit der Veränderung seiner Bestandteile, die wir eben kennengelernt haben. Theoretisch könnte ein Neuronales Netz also lernen, indem es

1. neue Verbindungen entwickelt,
2. vorhandene Verbindungen löscht,

3. Verbindungsgewichte verändert,
4. Schwellenwerte von Neuronen ändert,
5. eine oder mehrere der drei Neuronenfunktionen (wir erinnern uns: Aktivierungs-, Propagierungs- und Ausgabefunktion) abwandelt,
6. neue Neurone entwickelt
7. oder aber vorhandene Neurone löscht (und damit natürlich Verbindungen).

Wir behandeln die Gewichtsveränderung zunächst als die gängigste. Weiterhin kann das Löschen von Verbindungen hierdurch realisiert werden, dass man zusätzlich noch dafür Sorge trägt, dass eine zum Löschen auf 0 gesetzte Verbindung nicht weiter trainiert wird. Man kann weiterhin Verbindungen entwickeln durch das Setzen einer nicht vorhandenen Verbindung (welche ja in der Verbindungsmatrix den Wert 0 innehat) auf einen von 0 verschiedenen Wert. Für die Modifikation der Schwellenwerte verweise ich auf die Möglichkeit, diese als Gewichte zu implementieren (Abschnitt 3.4). Wir erschlagen also gleich vier der obigen Punkte durch reines Training von Verbindungsgewichten.

Die Veränderung von Neuronenfunktionen ist schwierig zu implementieren, nicht sehr intuitiv und auch nicht wirklich biologisch motiviert. Sie ist daher nicht verbreitet und wird hier zunächst nicht behandelt werden. Die Möglichkeiten, Neurone zu entwickeln oder zu löschen, liefern während des Trainings eines Neuronalen Netzes nicht nur gut eingestellte Gewichte, sondern optimieren auch noch die Netztopologie - sie gewinnen daher immer mehr an Interesse und werden oft mit evolutionären Verfahren realisiert. Da wir aber einsehen, dass wir einen Großteil der Lernmöglichkeiten bereits durch Gewichtsveränderungen abdecken können, sind auch sie zunächst nicht Gegenstand dieser Arbeit (es ist aber geplant, die Arbeit in diese Richtung zu erweitern).

SNIPE: Methoden der Klasse `NeuralNetwork` erlauben das Ändern von Verbindungsgewichten und Schwellenwerten sowie das Hinzufügen und Entfernen von Verbindungen sowie ganzen Neuronen. Methoden der Klasse `NeuralNetworkDescriptor` erlauben das Ändern der Aktivierungsfunktion pro Schicht.

Wir lassen also unser Neuronales Netz lernen, indem wir es die Verbindungsgewichte modifizieren lassen nach Regeln, die wir in Algorithmen fassen können – ein Lernverfahren ist also immer ein ***Algorithmus***, den wir einfach mithilfe einer Programmiersprache implementieren können. Ich werde später voraussetzen, dass wir definieren können, was eine *erwünschte, lernenswerte Ausgabe* ist (und an dieser stelle auch die *Trainingsbeispiele* formal definieren) und dass wir eine *Trainingsmenge* an Lernbeispielen besitzen. Eine Trainingsmenge sei folgendermaßen definiert:

Definition 4.1 (Trainingsmenge)**.** Als ***Trainingsmenge*** P bezeichnen wir eine Menge von Trainingsbeispielen, welche wir zum Training unseres Neuronales Netzes verwenden.

Ich stelle nun die drei wesentlichen Paradigmen des Lernens anhand der Unterschiede in der Beschaffenheit der Trainingsmenge vor.

4.1.1 Unüberwachtes Lernen gibt dem Netz nur Eingabemuster, aber keine Lernhilfen

Unüberwachtes Lernen (engl. ***unsupervised learning***) ist die biologisch plausibelste Methode, die aber nicht für alle Fragestellungen geeignet ist. Gegeben sind nur Eingabemuster; das Netz versucht, ähnliche Muster zu identifizieren und in ähnliche Kategorien zu klassifizieren.

Definition 4.2 (Unüberwachtes Lernen)**.** Die Trainingsmenge besteht nur aus ***Eingabemustern***, das Netz versucht selbst, Ähnlichkeiten herauszufinden und Musterklassen zu bilden.

Als bekanntes Beispiel sei wieder auf die *selbstorganisierenden Karten von Kohonen* (Kap. 10) verwiesen.

4.1.2 Bestärkendes Lernen gibt dem Netz Feedback, ob es sich gut oder schlecht verhält

Beim ***bestärkenden Lernen*** (engl. ***reinforcement learning***) wird dem Netz nach erfolgtem Durchlauf immerhin ein Wahrheits- oder reeller Wert geliefert, der definiert, ob das Ergebnis richtig oder falsch ist. Intuitiv ist klar, dass dieses Verfahren für viele Anwendungen zielstrebiger funktionieren sollte als unüberwachtes Lernen, erhält das Netz doch konkrete Anhaltspunkte zur Lösungsfindung.

Definition 4.3 (Bestärkendes Lernen)**.** Die Trainingsmenge besteht aus *Eingabemustern*, nach erfolgtem Durchlauf wird dem Netz ein Wert zurückgegeben, ob das Ergebnis falsch oder richtig war, u.U. noch *wie* falsch oder richtig es war.

4.1.3 Überwachtes Lernen hilft dem Netz mit Trainingsbeispielen und zugehörigen Lösungen

Beim ***überwachten Lernen*** (engl. ***supervised learning***) existiert eine Trainingsmenge von Eingabemustern sowie deren korrekte Ergebnisse in Form der genauen Aktivierung sämtlicher Ausgabeneurone. Für jedes in das Netz eingegebene Trainingsmuster kann so beispielsweise die Ausgabe direkt mit der korrekten Lösung verglichen werden und anhand der Differenz die Netzgewichtungen geändert werden. Ziel ist eine Veränderung der Gewichte dahingehend, dass das Netz nach dem Training nicht nur selbstständig Ein- und Ausgabemuster assoziieren, sondern bis dato unbekannte, ähnliche Eingabemuster einem plausiblen Ergebnis zuführen, also *generalisieren* kann.

Definition 4.4 (Überwachtes Lernen)**.** Die Trainingsmenge besteht aus *Eingabemustern mit jeweiliger korrekter Lösung*, so dass dem Netz nach Ausgabe ein genauer *Fehlervektor*[1] zurückgegeben werden kann.

Dieses Lernverfahren ist biologisch nicht immer plausibel, aber exorbitant zielgerichteter als die anderen und daher sehr praktikabel.

Wir möchten hier zunächst die überwachten Lernverfahren allgemein betrachten, welche sich innerhalb dieser Abeit zunächst an folgendes Schema halten:

Eingabe des Eingabemusters (Aktivierung der Eingabeneurone),

Vorwärtspropagierung der Eingabe durch das Netz, Erzeugung der Ausgabe,

Vergleich der Ausgabe mit der korrekten Ausgabe (*Teaching Input*), liefert Fehlervektor (Differenzvektor),

Verbesserungen des Netzes werden aufbauend auf den Fehlervektor berechnet.

Anwendung der Verbesserung um die vorher berechneten Werte.

4.1.4 Offline oder Online lernen?

Zu beachten ist, dass das Lernen ***offline*** erfolgen kann (eine Menge von Trainingsbeispielen wird präsentiert, danach werden die Gewichte verändert, der Gesamtfehler wird mit Hilfe einer Fehlerfunktion errechnet bzw. einfach aufkumuliert; näheres hierzu im Abschnitt 4.4) oder aber ***online*** (nach jedem präsentierten Beispiel werden die Gewichte verändert). Beides bietet Vor- und Nachteile, auf die wir bei den

1 Den Begriff des Fehlervektors werden wir in Abschnitt 4.2 noch definieren, wenn es an die mathematische Formalisierung des Lernens geht.

Lernverfahren nötigenfalls eingehen werden. Offline-Trainingsverfahren werden auch ***Batch-Trainingsverfahren*** genannt, da ein Stapel Ergebnisse auf einmal korrigiert wird. Einen solchen Trainingsabschnitt eines ganzen Stapels Trainingsbeispiele samt der zugehörigen Veränderung der Gewichtswerte nennt man ***Epoche***.

Definition 4.5 (Offline-Lernen)**.** Man gibt mehrere Trainingsbeispiele auf einmal in das Netz ein, kumuliert die Fehler auf, und lernt für alle Trainingsbeispiele gleichzeitig.

Definition 4.6 (Online-Lernen)**.** Man lernt direkt durch den Fehler eines jeden Trainingsbeispiels.

4.1.5 Fragen, über die man sich vor dem Lernen Gedanken machen sollte

Die Anwendung der Schemata setzt natürlich voraus, dass man sich vorher über einige Fragen Gedanken gemacht hat, die ich hier gewissermaßen als Checkliste einbringen und im Laufe der Arbeit sukzessive beantworten möchte, soweit möglich:

- ▷ Woher kommt die Lerneingabe und in welcher Form erhalten wir sie?
- ▷ Auf welche Weise muss man die Gewichte modifizieren, so dass man möglichst schnell und sicher lernt?
- ▷ Wie kann man objektiv messen, wie *erfolgreich* der Lernprozess ist?
- ▷ Kann man ermitteln, welches das „beste“ Lernverfahren ist?
- ▷ Kann man vorhersagen, ob ein Lernverfahren terminiert, ob es also nach endlicher Zeit einen optimalen Zustand erreicht oder z.B. zwischen verschiedenen Zuständen oszilliert?
- ▷ Wie wird das Gelernte im Netz gespeichert?
- ▷ Kann man verhindern, dass neu gelernte Muster alte erlernte Assoziationen wieder zerstören (das sog. ***Stabilitäts-Plastizitäts-Dilemma***)?

Wir werden feststellen, dass alle diese Fragen nicht allgemein beantwortet werden können, sondern für jedes Lernverfahren und jede Topologie von Netzwerk neu diskutiert werden müssen.

4.2 Trainingsmuster und Teaching Input

Bevor wir unsere erste Lernregel kennenlernen, muss der *Teaching Input* eingeführt werden. Im (hier vorliegenden) Falle des überwachten Lernens setzen wir voraus, dass eine Trainingsmenge aus Trainingsmustern und dazugehörigen richtigen Ausgabewerten vorliegt, die man nach erfolgtem Training an den Ausgabeneuronen sehen möchte. Diese Ausgabewerte werden, bis das Netz trainiert ist d.h. solange es falsche Ausgaben erzeugt, als sogenannter Teaching Input bezeichnet, und zwar für jedes Neuron einzeln. Für ein Neuron j mit fehlerhafter Ausgabe o_j ist t_j also die Bezeichnung für den Teaching Input, die richtige oder gewünschte Ausgabe zu einem Trainingsmuster p.

Definition 4.7 (Trainingsmuster)**.** Als ***Trainingsmuster*** bezeichnen wir einen Eingabevektor p mit Komponenten $p_1, p_2, \dots, p_n$, dessen gewünschte Ausgabe wir kennen. Indem wir das Trainingsmuster in das Netz eingeben, erhalten wir eine Ausgabe, die wir mit dem Teaching Input, also der gewünschten Ausgabe vergleichen. Die ***Menge der Trainingsmuster*** nennen wir P. Sie enthält eine endliche Anzahl geordneter Paare (p, t) von Trainingsmustern mit zugehörigem gewünschten Output.

Trainingsmuster heißen im englischen ***Pattern***, weswegen sie hier mit p bezeichnet werden. Sie besitzen in der Literatur und in der weiteren Arbeit viele Synonyme, wie z.B. Pattern, Trainingsbeispiel, Muster, usw.

Definition 4.8 (Teaching Input)**.** Sei j Ausgabeneuron. Der ***Teaching Input*** t_j ist definiert als der gewünschte, korrekte Wert, den j nach der Eingabe eines bestimmten Trainingsmusters ausgeben sollte. Analog zum Vektor p kann man auch Teaching Inputs $t_1, t_2, \dots, t_n$ der Neurone zu einem Vektor t zusammenfassen. t ist immer auf ein bestimmtes Trainingsmuster p bezogen und ist, wie oben schon gesagt, in der Menge P der Trainingsmuster enthalten.

SNIPE: Für die Trainingsdaten relevante Klassen befinden sich im Paket `training`. Die Klasse `TrainingSampleLesson` ist zum Speichern von Trainingsmustern und Teaching Inputs gedacht und erlaubt auch einfache Vorverarbeitung der daten.

Definition 4.9 (Fehlervektor)**.** Für mehrere Ausgabeneurone $\Omega_1, \Omega_2, \dots, \Omega_n$ wird die Differenz von Ausgabevektor und Teaching Input unter einem Trainingsbeispiel p

$$E_p = \begin{pmatrix} t_1 - y_1 \\ \vdots \\ t_n - y_n \end{pmatrix}$$

als ***Fehlervektor***, manchmal auch als ***Differenzvektor*** bezeichnet. Je nachdem, ob man offline oder online lernt, bezieht er sich auf ein bestimmtes Trainingsmuster, oder den auf bestimmte Weise normalisierten Fehler aus einer Menge von Trainingsmustern.

Ich fasse noch einmal kurz zusammen, was wir jetzt an diesbezüglichen Vektoren definiert haben. Es gibt einen

Eingabevektor x, der in das Neuronale Netz eingegeben werden kann. Das Neuronale Netz gibt dann je nach Netzart einen

Ausgabevektor y aus. Das

Trainingsbeispiel p ist im Grunde nichts weiter als ein Eingabevektor. Wir verwenden ihn nur zum Trainieren, weil wir den dazugehörigen

Teaching Input t kennen, der nichts anderes als der gewünschte Ausgabevektor zu dem Trainingsbeispiel ist. Der

Fehlervektor E_p ist die Differenz zwischen Teaching Input t und tatsächlicher Ausgabe y.

Was also x und y für den allgemeinen Betrieb des Netzes sind, sind p und t für das Training des Netzes – und während des Trainings versuchen wir, y möglichst nah an t heranzubringen. Noch ein Tip zur Nomenklatur. Wir haben die Ausgabewerte eines Neurons i mit o_i bezeichnet. Die Ausgabe eines Ausgabeneurons Ω heißt also o_Ω. Wir nennen aber Ausgabewerte des Netzes y_Ω. Diese Netzausgaben sind natürlich auch nur Neuronenausgaben, allerdings von Ausgabeneuronen. Insofern gilt

$$y_\Omega = o_\Omega.$$

4.3 Umgang mit Trainingsbeispielen

Wir haben gesehen, wie wir grundsätzlich lernen *können* und welche Schritte wir dafür durchführen müssen. Nun sollten wir noch die Wahl der Trainingsdaten und die Lernkurve betrachten. Insbesondere interessant nach erfolgtem Lernvorgang ist auch die Frage, ob das Netz vielleicht nur ***auswendig gelernt*** hat – also unsere Trainingsbeispiele recht exakt der richtigen Ausgabe zuführen kann, jedoch für sämtliche anderen Probleme derselben Klasse falsche Antworten liefert.

Angenommen, wir wollen das Netz eine Abbildung $\mathbb{R}^2 \to \mathbb{B}^1$ trainieren lassen, und die Trainingsbeispiele aus Abb. 4.1 auf der folgenden Seite ansetzen: Dann könnte es sein, dass das Netz zum Schluss exakt die farblich markierten Bereiche um die

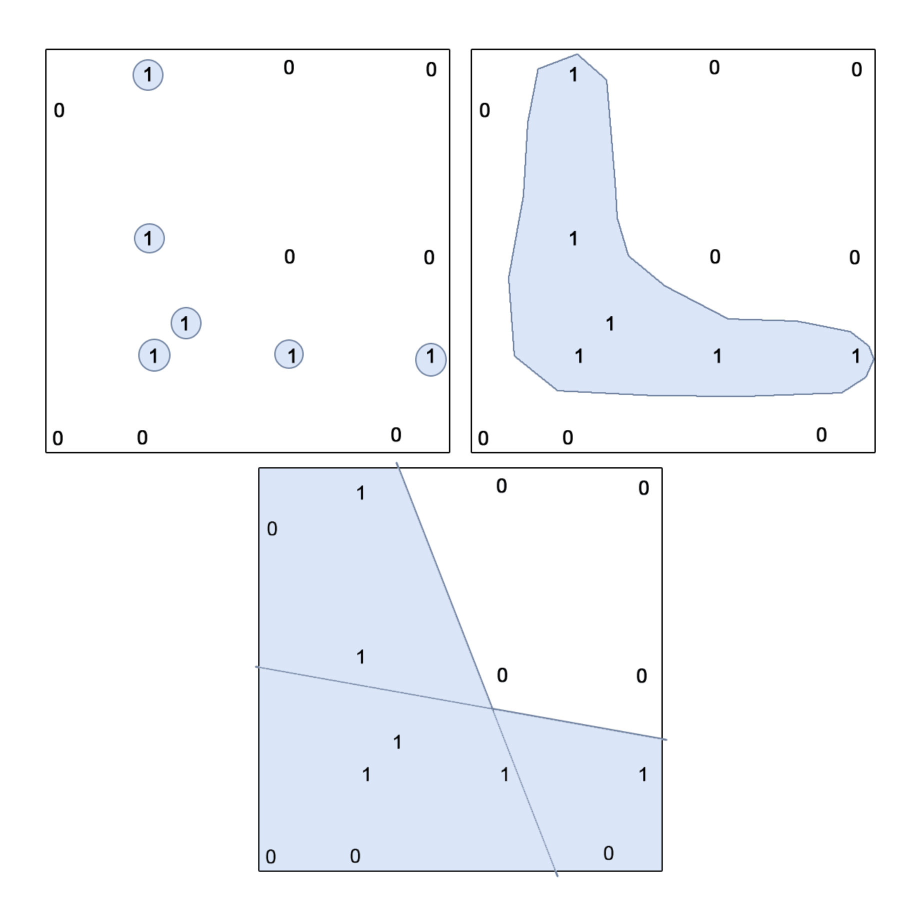


Abbildung 4.1: Veranschaulichung von Trainingsergebnissen derselben Trainingsmenge auf Netzen zu hoher (oben), richtiger (Mitte) oder zu niedriger Kapazität (unten).

Trainingsbeispiele herum mit der Ausgabe 1 markiert (Abb. 4.1 oben), und ansonsten überall 0 ausgibt – es hat also genug Speicherkapazität gehabt, sich auf die sechs Trainingsbeispiele mit einer 1 als Ausgabe zu konzentrieren, was auf ein zu großes Netz mit zu viel freier Speicherkapazität schließen lässt.

Andersherum kann auch ein Netz zu wenig Kapazität haben (Abb. 4.1 unten) – diese grobe Darstellung der Eingangsdaten entspricht auch nicht der guten Generalisierung, die wir uns wünschen. Es gilt also, hier den Mittelweg zu finden (Abb. 4.1 Mitte).

4.3.1 Es ist nützlich, die Menge der Trainingsbeispiele zu unterteilen

Ein Lösungsansatz für diese Probleme ist häufig, die Trainingsmenge zu *teilen*, und zwar

- in eine Trainingsmenge, mit der wir wirklich trainieren,
- und eine Testmenge, mit der wir unsere Fortschritte testen

– vorausgesetzt, wir haben ausreichend Trainingsbeispiele. Übliche Teilungsrelationen sind z.B. 70% für Trainingsdaten und 30% für Testdaten (zufällig gewählt). Wir können das Training beenden, wenn das Netz auf den Trainings- *und* Testdaten gute Werte liefert.

SNIPE: Die Methode `splitLesson` der Klasse `TrainingSampleLesson` erlaubt, eine TrainingSampleLesson anhand eines wählbaren Verhältnisses entzwei zu teilen.

Doch Achtung: Wenn man aufgrund eines schlechten Ergebnisses in den Testdaten an der Netzstruktur schraubt, bis auch diese gute Ergebnisse liefern, läuft man schnell Gefahr, das Netz auch auf die Testdaten zuzuschneiden, so dass diese zumindest indirekt auch ins Training eingehen, auch wenn sie nicht explizit für das Training genutzt werden. Abhilfe schafft ein dritter Validierungsdatensatz, der *nach* dem mutmaßlich erfolgreichen Training zum reinen Validieren benutzt wird.

Offensichtlich enthalten wir dem Neuronalen Netz dadurch, dass wir weniger Muster trainieren, Information vor und laufen damit Gefahr, dass es schlechter lernt. Es geht uns hier aber nicht um das 100% exakte Reproduzieren gegebener Beispiele, sondern um erfolgreiches Generalisieren und um Approximation einer ganzen Funktion – wofür es durchaus von Nutzen sein kann, weniger Informationen in das Netz zu trainieren.

4.3.2 Reihenfolgen der Musterpräsentation

Man kann auch verschiedene Strategien finden, in welcher Reihenfolge man Muster präsentiert: Präsentiert man sie zufällig, so gibt es keine Garantie, dass die Muster gleichverteilt erlernt werden (dennoch ist dies die gängigste Methode). Immer dieselbe Musterreihenfolge hingegen provoziert z.B. bei rekurrenten Netzen (später mehr dazu) ein Auswendiglernen der Muster. Abhilfe gegen beide Probleme würde hier eine *zufällige Permutation* schaffen, die aber – wie schon erwähnt – aufwändig zu berechnen ist.

SNIPE: Die Methode `shuffleSamples` der Klasse `TrainingSampleLesson` permutiert die Lesson.

4.4 Lernkurve und Fehlermessung

Die Lernkurve beschreibt den zeitlichen Verlauf des Fehlers, der auf verschiedene Weisen ermittelt werden kann – die Motivation für die Erschaffung einer Lernkurve liegt darin, dass man mit ihr darstellen kann, ob das Netz Fortschritte macht oder nicht. Der Fehler sollte hierbei normiert sein, also ein Abstandsmaß zwischen richtigem und aktuellem Output des Netzes darstellen. Beispielsweise können wir den musterspezifischen, quadratischen Fehler mit Vorfaktor nehmen, wie wir ihn für die Herleitung von Backpropagation of Error verwenden werden (Seien Ω Outputneurone und O die Menge derselben.):

$$\mathrm{Err}_p = \frac{1}{2} \sum_{\Omega \in O} (t_\Omega - y_\Omega)^2 \tag{4.1}$$

Definition 4.10 (Spezifischer Fehler)**.** Der ***spezifische Fehler*** Err_p wird über ein einziges Trainingsbeispiel, also online, gebildet.

Weiter üblich sind der ***Root-Mean-Square*** (kurz: ***RMS***) sowie der ***Euklidische Abstand***.

Der Euklidische Abstand (Verallgemeinerung des Satzes des Pythagoras) ist gut für niedere Dimensionen, wo wir uns seinen Nutzen auch noch bildlich vorstellen können.

Definition 4.11 (Euklidischer Abstand)**.** Der Euklidische Abstand zweier Vektoren t und y ist definiert zu

$$\mathrm{Err}_p = \sqrt{\sum_{\Omega \in O} (t_\Omega - y_\Omega)^2}. \tag{4.2}$$

Der Root-Mean-Square wird allgemein häufig verwendet, weil er auf grobe Ausreißer mehr Rücksicht nimmt.

Definition 4.12 (Root-Mean-Square)**.** Der Root-Mean-Square zweier Vektoren t und y ist definiert zu

$$\text{Err}_p = \sqrt{\frac{\sum_{\Omega \in O}(t_\Omega - y_\Omega)^2}{|O|}}. \tag{4.3}$$

Für Offline-Lernen ist auch der gesamte Fehler über eine Trainingsepoche von Interesse und Nutzen:

$$\text{Err} = \sum_{p \in P} \text{Err}_p \tag{4.4}$$

Definition 4.13 (Gesamtfehler)**.** Der ***Gesamtfehler*** Err wird über alle Trainingsbeispiele, also offline, gebildet.

Analog können wir einen Gesamt-RMS und einen Gesamt-Euklidischen Abstand über eine ganze Epoche bilden. Natürlich sind auch andere Fehlermaße als das euklidische oder der RMS denkbar. Für weitere Varianten der Fehlermessungen kann ich nur raten, einen Blick in den Technical Report von Prechelt [Pre94] zu werfen – hier werden sowohl Fehlermaße als auch Beispielproblemstellungen intensiv diskutiert (darum kommt dazu später auch noch einmal eine ähnliche Anmerkung im Rahmen der Diskussion von Beispielproblemen).

SNIPE: In der Klasse `ErrorMeasurement` befinden sich verschiedene statische Methoden, welche verschiedene Verfahren der Fehlermessung implementieren.

Abhängig von unserem Fehlermessverfahren sieht natürlich unsere Lernkurve auch anders aus. Eine ideale Lernkurve sieht aus wie eine negative Exponentialfunktion, ist also proportional zu e^{-t} (Abb. 4.2 auf der folgenden Seite). Insofern stellt man die Lernkurve am anschaulichsten mit einer logarithmischen Skala dar (Abb. 4.2, zweites Diagramm von unten) – bei dieser Skalierungskombination bedeutet eine absinkende Gerade einen exponentiellen Abfall des Fehlers.

Bei guter Arbeit des Netzes, einfachen Problemstellungen und logarithmischer Darstellung von Err sieht man also bildlich gesprochen eine absinkende Gerade, die unten oft „Zacken" bildet – hier stoßen wir an das Auflösungsvermögen unserer 64-Bit-Darstellung im Computer und haben tatsächlich das Optimum dessen erlernt, was unser Netz lernen kann.

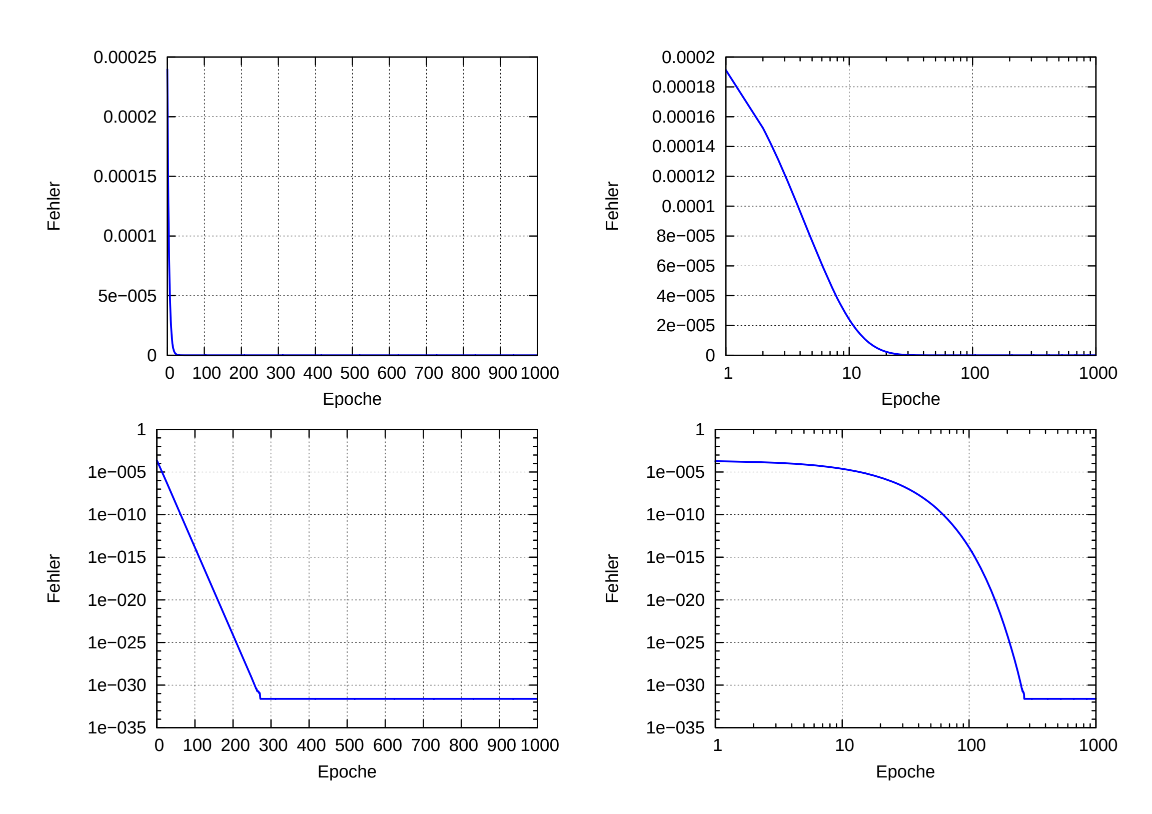


Abbildung 4.2: Alle vier Abbildungen stellen dieselbe (idealisierte, da sehr glatte) Lernkurve dar. Achten Sie auf die wechselnden logarithmischen und linearen Skalierungen! Beachten Sie auch den kleinen „Ungenauigkeits-Zacken“, sichtbar am Knick der Kurve im ersten und zweiten Diagramm von unten.

Typische Lernkurven können auch einige wenige flache Bereiche aufweisen, also Stufen beschreiben, dies ist kein Zeichen für einen schlecht funktionierenden Lernvorgang. Wie wir weiterhin an der Abbildung 4.2 sehen, kann man mit geeigneter Darstellung fast jede nur etwas sinkende Lernkurve schön aussehen lassen – insofern sollte man beim Lesen der Literatur etwas vorsichtig sein.

4.4.1 Wann hört man auf zu lernen?

Nun ist die große Frage: Wann hört man auf zu lernen? Üblicherweise hört das Training auf, wenn derjenige, der vor dem trainierenden Computer sitzt, das „Gefühl“ hat, der Fehler wäre gering genug. In der Tat gibt es dafür wie so oft keine Patentlösung und so kann ich hier wieder nur Denkansätze geben, welche allerdings alle für eine objektivere Sicht auf das Vergleichen mehrerer Lernkurven setzen.

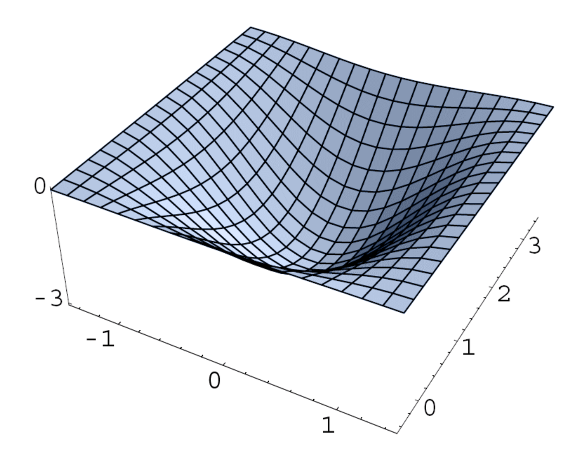
Es stärkt beispielsweise das Vertrauen in ein Ergebnis, wenn das Netz für mehrere verschiedene zufällige Initialisierungen immer ungefähr die gleiche Endfehlerrate erreicht – mehrmals initialisieren und trainieren macht das Ergebnis also ein wenig objektiver.

Auf der anderen Seite kann sich auch ein Bild ergeben, bei dem eine anfangs schneller fallende Kurve nach längerem Lernen von einer anderen Kurve noch überholt wird: Dies kann darauf hinweisen, dass entweder die Lernrate der schlechteren Kurve zu hoch war oder aber die schlechtere einfach in einem Nebenminimum hängen geblieben ist, dieses aber schneller gefunden hat.

Noch einmal zur Erinnerung: Größere Fehlerwerte sind schlechter.

Auf jeden Fall sei aber beachtet: Worauf sich viele beim Lernen verlassen (und sich danach wundern, dass wenig funktioniert), ist, die Lernkurve nur in Bezug auf die Trainingsdaten zu bilden – man sollte also der Objektivität halber nicht vergessen, die Testdaten der Übersichtlichkeit halber auf eine zweite Lernkurve aufzutragen, die üblicherweise etwas schlechtere Werte liefert und auch stärker oszilliert, aber bei guter Generalisierung auch sinkt.

Wenn das Netz dann irgendwann anfängt, die Beispiele auswendig zu lernen, kann man so durch die Lernkurve Hinweise darauf erhalten: Wenn die Lernkurve der Testbeispiele plötzlich rapide steigt, während die Lernkurve für die Trainingsdaten weiter sinkt, kann dies ein Indikator für Auswendiglernen und schlechter werdende Generalisierung sein. Hier könnte man dann entscheiden, ob das Netz am nächsten Punkt der beiden Kurven bereits gut genug gelernt hat und der Endzeitpunkt des Lernens vielleicht hier anzusetzen ist (dieses Verfahren wird ***Early Stopping*** genannt).



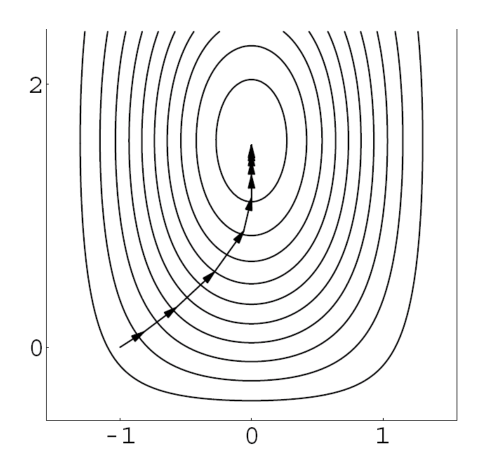


Abbildung 4.3: Veranschaulichung des Gradientenabstiegs auf zweidimensionaler Fehlerfunktion. Wir gehen entgegengesetzt von g, also mit dem steilsten Abstieg einem Tiefpunkt entgegen, wobei die Schrittweite proportional zu $|g|$ ist (je steiler der Abstieg, desto größer die Schrittweite). Links ist die Fläche in 3D gezeigt, rechts die Schritte über die Höhenlinien in 2D. Hier wird ersichtlich, wie eine Bewegung in Gegenrichtung von g in Richtung Minimum der Funktion erfolgt und proportional zu $|g|$ ständig langsamer wird. **Quelle**: `http://webster.fhs-hagenberg.ac.at/staff/sdreisei/Teaching/WS2001-2002/PatternClassification/graddescent.pdf`

Ich weise noch einmal darauf hin, dass dies alles nur *Indikatoren* und keine Wenn-Dann-Schlüsse sind.

4.5 Gradientenbasierte Optimierungsverfahren

Um die mathematische Grundlage für einige der folgenden Lernverfahren zu schaffen, möchte ich zunächst kurz erklären, was man unter einem *Gradientenabstieg* versteht – das Lernverfahren *Backpropagation of Error* beispielsweise baut auf diesen mathematischen Grundlagen auf und erbt so die Vor- und Nachteile des Gradientenabstiegs.

Gradientenabstiegsverfahren werden im Allgemeinen verwendet, um Maxima oder Minima n-dimensionaler Funktionen auszumachen. In der Illustration (Abb. 4.3) beschränke ich mich übersichtlicherweise auf zwei Dimensionen, der Dimensionsanzahl sind aber prinzipiell keine Grenzen gesetzt.

Hierbei ist der *Gradient* ein Vektor g, der für jeden differenzierbaren Punkt einer Funktion definiert ist, genau in die Richtung des *steilsten Anstiegs* von diesem Punkt aus deutet und durch seinen Betrag $|g|$ den Steigungsgrad in diese Richtung angibt. Der Gradient ist also die *Verallgemeinerung der Ableitung für mehrdimensionale Funktionen*. Folglich deutet der *negative Gradient* $-g$ genau in die Richtung des *steilsten Abstiegs*. Der Operator für einen Gradienten ∇ wird als ***Nabla-Operator*** bezeichnet, die Gesamtschreibweise für den Gradienten g des Punktes (x, y) einer zweidimensionalen Funktion f lautet dabei z.B. $g(x, y) = \nabla f(x, y)$.

Definition 4.14 (Gradient)**.** Sei g ***Gradient***. Dann ist g ein n-komponentiger Vektor, der für jeden Punkt einer (differenzierbaren) n-dimensionalen Funktion $f(x_1, x_2, \ldots, x_n)$ bestimmt ist. Die Operatorschreibweise für den Gradienten ist definiert als

$$g(x_1, x_2, \ldots, x_n) = \nabla f(x_1, x_2, \ldots, x_n)$$

g zeigt für jeden Punkt von f in Richtung des stärksten Anstiegs von diesem Punkt aus, wobei $|g|$ dem Grad dieser Steigung entspricht.

Als *Gradientenabstieg* bezeichnen wir, von beliebigem Startpunkt unserer Funktion aus entgegen dem Gradienten g schrittweise *bergab* zu gehen (anschaulich gesprochen in die Richtung, in die auch eine Kugel vom Startpunkt aus rollen würde), wobei die Schrittgröße proportional zu $|g|$ ist. Auf flachen Plateaus bewegen wir uns also langsam, bei großer Steigung schnell den steilsten Weg hinab. Geraten wir in ein Tal, so werden wir es je nach Größe unserer Schritte überspringen oder auf dem gegenüberliegenden Hang wieder ins Tal umkehren, um durch hin- und hergehen dem tiefsten Punkt des Tals immer näher zu kommen, ähnlich der Bewegung unserer Kugel innerhalb einer runden Schüssel.

Definition 4.15 (Gradientenabstieg)**.** Sei f eine n-dimensionale Funktion und $s = (s_1, s_2, \ldots, s_n)$ gegebener Startpunkt. Als ***Gradientenabstieg*** bezeichnen wir, von $f(s)$ aus entgegen der Richtung von g, also in Richtung von $-g$ mit Schritten in Größe von $|g|$ in Richtung immer kleinerer Werte von f zu gehen.

Gradientenabstiegsverfahren sind kein fehlerfreies Optimierungsverfahren (wie wir in den nächsten Abschnitten sehen werden), aber sie funktionieren doch so gut, dass sie in der Praxis häufig eingesetzt werden. Dennoch wollen wir uns ihre potenziellen Nachteile kurz vor Augen führen.

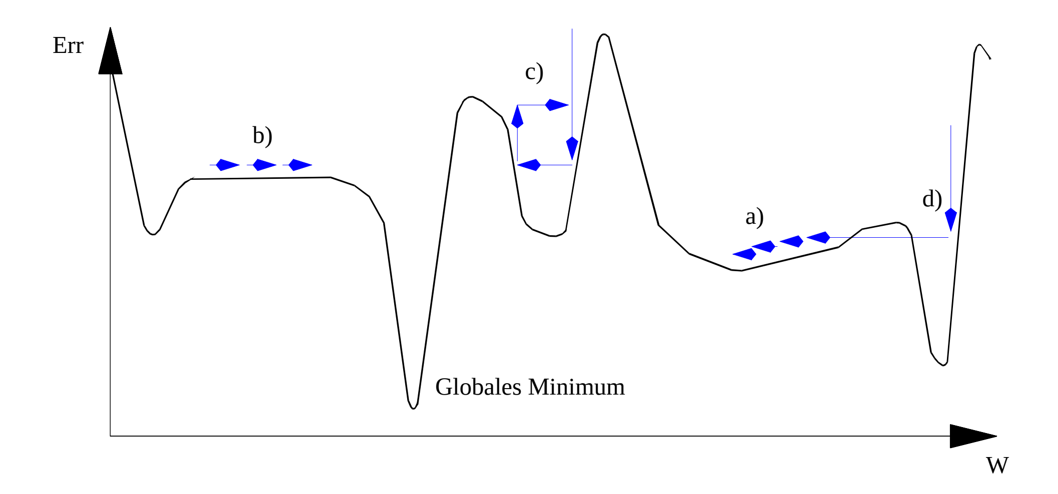


Abbildung 4.4: Mögliche Fehler während eines Gradientenabstiegs: a) Finden schlechter Minima, b) Quasi-Stillstand bei kleinem Gradienten, c) Oszillation in Schluchten, d) Verlassen guter Minima.

4.5.1 Gradientenverfahren bringen verschiedene Probleme mit sich

Wie in Abschnitt 4.5 angedeutet, ist der Gradientenabstieg (und damit Backpropagation) erfolgversprechend, jedoch nicht fehlerresistent, wobei eines der ***Probleme*** ist, dass man nicht immer anhand des Ergebnisses ersehen kann, ob ein Fehler passiert ist.

4.5.1.1 Häufig konvergieren Gradientenverfahren nur gegen suboptimale Minima

Jedes Gradientenabstiegsverfahren kann zum Beispiel in einem lokalen Minimum hängen bleiben (ein Beispiel findet sich in Teil a der Abb. 4.4) – dieses Problem wächst mit der Größe der Fehlerfläche an und hierfür gibt es keine allgemeingültige Lösung. In der Realität kann man nicht wissen, ob man das optimale Minimum gefunden hat – also gibt man sich zufrieden, sobald man ein Minimum ausreichender Qualität gefunden hat.

4.5.1.2 Flache Plateaus in der Fehleroberfläche können das Training sehr verlangsamen

Auch wird der Gradient beispielsweise beim Durchlaufen eines flachen Plateaus verschwindend klein (es ist eben kaum Steigung vorhanden (Teil b der Abb. 4.4), was sehr viele weitere Schritte nötig macht. Ein theoretisch möglicher Gradient von 0 würde den Abstieg gar ganz zum Stillstand bringen.

4.5.1.3 Gute Minima können wieder verlassen werden

Auf der anderen Seite ist der Gradient an einem steilen Hang sehr groß, so dass man große Schritte macht und u.U. ein gutes Minimum übersieht (Teil d der Abb. 4.4).

4.5.1.4 Steile Schluchten in der Fehlerfunktion können Oszillationen hervorrufen

Ein plötzlicher Wechsel von einem sehr stark negativen zu einem sehr stark positiven Gradienten kann sogar zu einer Oszillation führen (Teil c der Abb. 4.4). An und für sich hört man von diesem Fehler in der Natur selten, so dass wir uns über Möglichkeiten b und d Gedanken machen können.

4.6 Beispielproblemstellungen sind nützlich, um das selbst programmierte Netz und Lernverfahren zu testen

Wir haben nun das Lernen noch nicht sehr, aber zumindest ein wenig von der formalen Seite betrachtet – nun ist es an der Zeit, dass ich ein paar Beispielprobleme vorstelle, mit denen man sehr gut ausprobieren kann, ob ein implementiertes Netz und ein Lernverfahren korrekt arbeiten.

4.6.1 Boolesche Funktionen

Gerne wird als Beispiel das genommen, was in den 1960er Jahren nicht ging: Die XOR-Funktion ($\mathbb{B}^2 \to \mathbb{B}^1$), welches wir noch ausführlich besprechen werden. Trivial erwarten wir hier die Ausgaben 1.0 bzw. -1.0 je nachdem , ob die Funktion XOR 1 oder 0 ausgibt – und genau hier liegt der erste Anfängerfehler, den man machen kann.

i_1	i_2	i_3	Ω
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Tabelle 4.1: Darstellung der Paritätsfunktion mit drei Eingaben.

Für Ausgaben nahe 1 oder -1, also nahe den Grenzwerten des Tangens Hyperbolicus (bzw. im Falle der Fermifunktion 0 oder 1), benötigt das Netz sehr große Netzeingaben. Die einzige Chance, diese Netzeingaben zu erreichen, ist durch große Gewichte, die erlernt werden müssen: Der Lernvorgang wird sehr verlängert. Es ist also klüger, als gewünschte Ausgaben in die Teaching Inputs 0.9 bzw. -0.9 einzugeben oder aber zufrieden zu sein, wenn das Netz diese anstatt 1 und -1 ausgibt.

Beliebt als Beispiel für Singlelayerperceptrons sind auch die Booleschen Funktionen AND und OR.

4.6.2 Die Paritätsfunktion

Die Paritätsfunktion bildet eine Menge von Bits auf 1 oder 0 ab, je nachdem, ob eine gerade Anzahl Inputbits auf 1 gesetzt ist oder nicht – es handelt sich also grundsätzlich um eine Funktion $\mathbb{B}^n \to \mathbb{B}^1$. Sie ist durch leichte Lernbarkeit bis ca. $n = 3$ gekennzeichnet (dargestellt in Tab. 4.1), der Lernaufwand steigt aber ab $n = 4$ rapide an. Der Leser möge doch einmal eine Wertetabelle für die 2-bit-Paritätsfunktion erstellen – was fällt auf?

4.6.3 Das 2-Spiralen-Problem

Nehmen wir als Trainingsbeispiel für eine Funktion zwei ineinander gewundene Spiralen (Abb. 4.5 auf der rechten Seite), wobei die Funktion natürlich eine Abbildung $\mathbb{R}^2 \to \mathbb{B}^1$ repräsentiert. Eine der Spiralen ist mit dem Outputwert 1 belegt, die andere mit 0. Hier

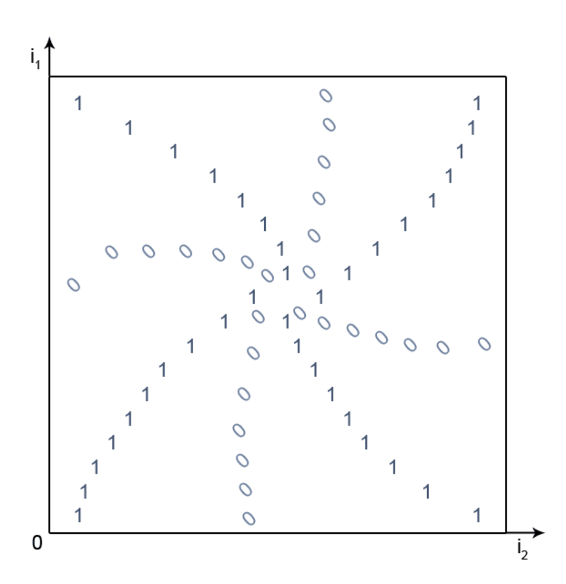


Abbildung 4.5: Skizze zum Trainingsbeispiel des 2-Spiralen-Problems

hilft Auswendiglernen nicht groß weiter, das Netz muss wirklich die Abbildung an sich verstehen. Auch dieses Beispiel kann mit einem MLP gelöst werden.

4.6.4 Das Schachbrettproblem

Wir kreieren uns wieder eine zweidimensionale Funktion der Form $\mathbb{R}^2 \to \mathbb{B}^1$ und geben schachbrettartige Trainingsbeispiele an (Abb. 4.6 auf der folgenden Seite), wobei ein eingefärbtes Feld eine 1 repräsentiert, alle anderen 0. Hier steigt die Schwierigkeit wieder mit Größe der Funktion: Während ein 3×3-Feld noch leicht zu lernen ist, sieht es mit größeren Feldern schon schwierig aus.

Das 2-Spiralen-Problem ist dem Schachbrettproblem sehr ähnlich, nur dass bei ersterem mathematisch gesehen Polarkoordinaten statt kartesischen Koordinaten verwendet werden. Eine letzte Kleinigkeit möchte ich noch als Beispiel vorstellen: Die Identität.

4.6.5 Die Identitätsfunktion

Mit linearen Aktivierungsfunktionen wäre die Identitätsabbildung von $\mathbb{R}^1$ nach $\mathbb{R}^1$ (natürlich fairerweise im Wertebereich der verwendeten Aktivierungsfunktion) für das

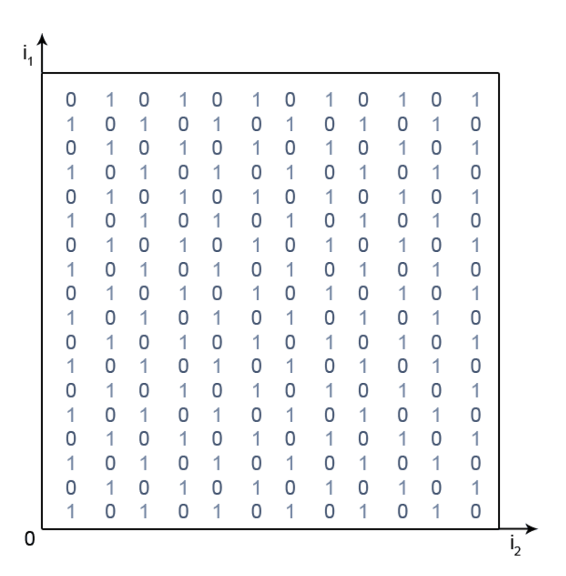


Abbildung 4.6: Skizze von Trainingsbeispielen des Schachbrettproblems

Netz kein Problem, doch wir legen ihm durch Verwendung unserer sigmoiden Funktionen Steine in den Weg, so dass es für das Netz sehr schwierig ist, die Identität zu lernen. Versuchen Sie es doch einmal spaßeshalber.

4.6.6 Es gibt eine Vielzahl von Beispielproblemstellungen

Für die Einarbeitung in weitere beispielhafte Problemstellungen möchte ich ausdrücklich den Technical Report von Prechelt [Pre94] ans Herz legen, der auch schon im Rahmen der Fehlermessungsverfahren erwähnt wurde.

Nun können wir unsere erste mathematische Lernregel betrachten.

4.7 Die Hebbsche Lernregel ist der Grundstein für die meisten anderen Lernregeln

Die 1949 von Donald O. Hebb formulierte ***Hebbsche Lernregel*** [Heb49] bildet den Grundstein für die meisten komplizierteren Lernregeln, welche wir in dieser Arbeit

besprechen. Hierbei wird unterschieden zwischen der Urform und der allgemeineren Form, einer Art Grundgerüst für viele andere Lernregeln.

4.7.1 Urform

Definition 4.16 (Hebbsche Lernregel). „Wenn Neuron j eine Eingabe von Neuron i erhält und beide gleichzeitig stark aktiv sind, dann erhöhe das Gewicht $w_{i,j}$ (also die Stärke der Verbindung von i nach j).“ Mathematisch ausgedrückt lautet sie also:

$$\Delta w_{i,j} \sim \eta o_i a_j \tag{4.5}$$

wobei $\Delta w_{i,j}$ die ***Änderung des Gewichtes*** von i nach j bezeichnet, welche proportional zu folgenden Faktoren ist:

- der Ausgabe o_i des Vorgängerneurons i,
- der Aktivierung a_j des Nachfolgerneurons j,
- sowie einer Konstante η, der Lernrate, auf die wir in Abschnitt 5.4.3 noch genau eingehen.

Gewichtsänderungen $\Delta w_{i,j}$ werden einfach auf das Gewicht $w_{i,j}$ aufaddiert.

Warum spreche ich in der umgangssprachlichen Formulierung zweimal von *Aktivierung*, schreibe jedoch in der Formel von o_i und a_j, also von der *Ausgabe* des Neurons i und der Aktivierung des Neurons j? Wir erinnern uns, dass sehr oft die Identität als Ausgabefunktion verwendet wird und so a_i und o_i eines Neurons oft identisch sind – weiterhin postulierte Hebb seine Lernregel weit vor der Spezifikation technischer Neurone. Wenn man bedenkt, dass diese Lernregel gerne bei binären Aktivierungen verwendet wurde, ist klar, dass die Gewichte bei möglichen Aktivierungen $(1, 0)$ entweder anwachsen oder gleichbleiben. Sie würden also über kurz oder lang ins unendliche gehen, da sie bei Fehlern nur „nach oben“ korrigiert werden können. Dies kann ausgeglichen werden, indem man die Aktivierungen $(-1, 1)$ verwendet[2]. So werden die Gewichte bei Nichtübereinstimmung der Aktivierung von Vorgänger- und Nachfolgerneuron verringert, sonst verstärkt.

2 Das ist dann aber nicht mehr die „Originalversion“ der Hebbschen Lernregel.

4.7.2 Verallgemeinerte Form

Die meisten weiteren hier besprochenen Lernregeln sind eine Spezialisierung der mathematisch allgemeineren Form [MR86] der Hebbschen Lernregel.

Definition 4.17 (Hebb-Regel, allgemeiner)**.** Die ***verallgemeinerte Form der Hebbschen Regel*** benennt nur die Proportionalität der Gewichtsänderung zum Produkt zweier nicht näher definierter Funktionen, allerdings mit definierten Eingabewerten.

$$\Delta w_{i,j} = \eta \cdot h(o_i, w_{i,j}) \cdot g(a_j, t_j) \tag{4.6}$$

Hierbei ergibt das Produkt der Funktionen

- ▷ $g(a_j, t_j)$ und
- ▷ $h(o_i, w_{i,j})$
- ▷ sowie wieder der konstanten Lernrate η

die Gewichtsänderung. Wie man sieht, nimmt h also die Ausgabe der Vorgängerzelle o_i sowie das Gewicht von Vorgänger zu Nachfolger $w_{i,j}$ entgegen, während g die tatsächliche und gewünschte Aktivierung des Nachfolgers a_j sowie t_j (t steht hier für den erwähnten *Teaching Input*) erwartet. In dieser allgemeinen Definition sind g und h, wie schon angemerkt, nicht konkretisiert – wir werden uns daher jetzt auf den vor Gleichung 4.6 angesprochenen Pfad der Spezialisierung begeben und unser erstes Netzparadigma samt Lernverfahren kennenlernen, nachdem wir nun eine kurze Ansicht dessen gehabt haben, wie eine Lernregel aussehen kann und uns Gedanken zum Lernen an sich gemacht haben.

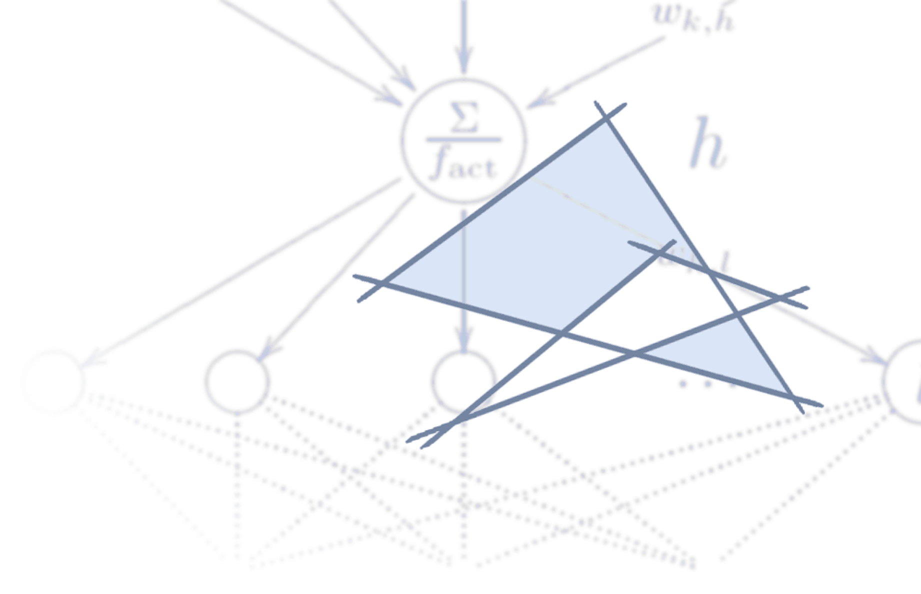
Übungsaufgaben

Aufgabe 7. Berechnen Sie für die folgenden Datenpunkte den Mittelwert μ und die Standardabweichung σ.

$$\begin{aligned} p1 &= (2, 2, 2) \\ p2 &= (3, 3, 3) \\ p3 &= (4, 4, 4) \\ p4 &= (6, 0, 0) \\ p5 &= (0, 6, 0) \\ p6 &= (0, 0, 6) \end{aligned}$$

Teil II

Überwacht lernende Netzparadigmen



Kapitel 5

Das Perceptron, Backpropagation und seine Varianten

Der Klassiker unter den Neuronalen Netzen. Wenn von einem Neuronalen Netz gesprochen wird, ist meistens ein Perceptron oder eine Variation davon gemeint. Perceptrons sind mehrschichtige Netze ohne Rückkopplung, mit festen Eingabe- und Ausgabeschichten. Beschreibung des Perceptrons, seiner Grenzen und seiner Erweiterungen, welche die Grenzen umgehen sollen. Herleitung von Verfahren, es lernen zu lassen, und Diskussion über deren Probleme.

Wie schon in der Geschichte Neuronaler Netze erwähnt, wurde das Perceptron von Frank Rosenblatt 1958 beschrieben [Ros58]. Rosenblatt legte als Komponenten des Perceptrons zunächst die schon besprochene *gewichtete Summe*, sowie eine nichtlineare Aktivierungsfunktion fest.

Obwohl es keine wirklich feste Definition des Perceptrons gibt, ist meistens ein *FeedForward-Netz mit ShortCut-Connections* gemeint, das eine Schicht von Abtastneuronen (*Retina*) mit *statisch* gewichteten Verbindungen zur nächsten Schicht besitzt, die wir Eingabeschicht nennen (Abb. 5.1 auf Seite 91); alle Gewichte ab der Eingabeschicht dürfen aber verändert werden. Die der Retina nachgeordneten Neuronen stellen Musterdetektoren dar. Wir verwenden hier zunächst ein *binäres Perceptron*, bei dem jedem Outputneuron genau zwei mögliche Ausgabewerte zur Verfügung stehen (z.B. $\{0, 1\}$ oder $\{-1, 1\}$). Wir verwenden also eine binäre Schwellenwertfunktion als Aktivierungsfunktion, abhängig vom jeweiligen Schwellenwert Θ des Outputneurons.

Gewissermaßen stellt eine binäre Aktivierungsfunktion also eine IF-Abfrage dar, die man durch negative Gewichte auch negieren kann – man kann also mit dem Perceptron wirkliche logische Informationsverarbeitung durchführen.

Ob das sinnvoll ist, sei einmal dahingestellt – Boolesche Logik kann man natürlich auch einfacher haben. Ich möchte nur darstellen, dass Perceptrons durchaus als simple logische Bausteine nutzbar sind und man mit geschickten hintereinander- und zusammengeschalteten Perceptrons theoretisch jede Boolesche Funktion realisieren kann. Wir werden aber noch sehen, dass dies ohne Hintereinanderschaltung nicht möglich ist. Bevor wir das Perceptron an sich definieren, möchte ich zunächst einige Neuronenarten, welche wir in diesem Kapitel verwenden werden, definieren.

Definition 5.1 (Eingabeneuron)**.** Ein ***Eingabeneuron*** (auch ***Inputneuron*** genannt) ist ein ***Identitätsneuron*** – es gibt genau das weiter, was es als Eingabe erhält. Es repräsentiert also die Identitätsfunktion, die wir durch das Symbol ⁄ andeuten wollen. Wir stellen ein Eingabeneuron daher mit dem Symbol (⁄) dar.

Definition 5.2 (Informationsverarbeitendes Neuron)**.** ***Informationsverarbeitende Neurone*** verarbeiten die eingegebene Information auf irgendeine Weise, repräsentieren also nicht die Identitätsfunktion. Ein ***Binäres Neuron*** summiert alle Eingaben, die es erhält, durch die gewichtete Summe als Propagierungsfunktion auf, was wir mit dem Summenzeichen Σ skizzieren wollen. Die Aktivierungsfunktion des Neurons ist dann die binäre Schwellenwertfunktion, die mit ⎍ skizziert werden kann. Dies bringt uns insgesamt zu der Symboldarstellung ($\frac{\Sigma}{⎍}$). Analog werden andere Neurone mit gewichteter Summe als Propagierungsfunktion, jedoch den Aktivierungsfunktionen *Tangens Hyperbolicus*, *Fermifunktion* oder einer separat definierten Aktivierungsfunktion f_{act} als

$$\frac{\Sigma}{\text{Tanh}} \qquad \frac{\Sigma}{\text{Fermi}} \qquad \frac{\Sigma}{f_{\text{act}}}$$

dargestellt. Diese Neurone bezeichnen wir dann z.B. auch als ***Fermi-Neuron*** oder ***Tanh-Neuron***.

Ausgabeneurone werden auch oft ***Outputneuron*** genannt. Nachdem wir nun die Bestandteile eines Perceptrons kennen, können wir es auch definieren.

Definition 5.3 (Perceptron)**.** Das ***Perceptron*** (Abb. 5.1 auf der rechten Seite) ist[1] ein FeedForward-Netz, in welchem es eine ***Retina*** gibt, die der reinen Datenaufnahme

1 Es mag dem einen oder anderen Leser sauer aufstoßen, dass ich behaupte, es gäbe keine Definition für ein Perceptron, das Perceptron aber im nächsten Absatz definiere. Ich schlage daher vor, meine Definition im Hinterkopf zu behalten, aber nur für diese Arbeit als wirklich gegeben anzusehen.

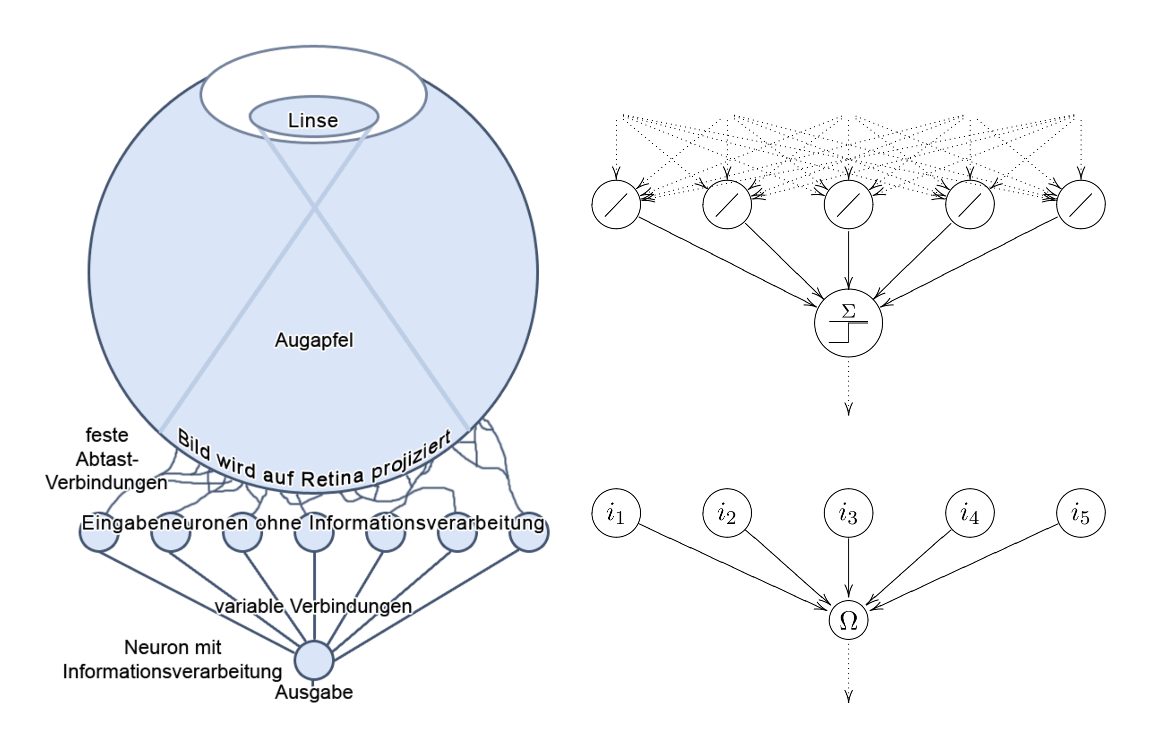


Abbildung 5.1: Aufbau eines Perceptrons mit einer Schicht variabler Verbindungen in verschiedenen Ansichten. Die durchgezogene Gewichtsschicht in den unteren beiden Abbildungen ist trainierbar.
Links: Am Beispiel der Informationsabtastung im Auge.
Rechts oben: Skizze desselben mit eingezeichneter fester Gewichtsschicht unter Verwendung der definierten funktionsbeschreibenden Designs für Neurone.
Rechts unten: Ohne eingezeichnete feste Gewichtsschicht, mit Benennung der einzelnen Neuronen nach unserer Konvention. Wir werden die feste Gewichtschicht im weiteren Verlauf der Arbeit nicht mehr betrachten.

dient und fest gewichtete Verbindungen zur ersten Neuronenschicht (Eingabeschicht) besitzt. Der festen Gewichtsschicht folgt mindestens eine trainierbare Gewichtsschicht. Eine Neuronenschicht ist zur jeweils nächsten vollverknüpft. Die erste Schicht des Perceptrons besteht aus den oben definierten *Eingabeneuronen*.

Oft enthält ein FeedForward-Netz auch ShortCuts, was aber nicht exakt der ursprünglichen Beschreibung entspricht und daher hier auch nicht der Definition hinzugefügt wird. Wir sehen, dass die Retina gar nicht in den unteren Teil der Abb. 5.1 mit einbezogen wird – in der Tat wird meist (vereinfachend und für die Implementierung ausreichend) die erste Neuronenschicht nach der Retina als Eingabeschicht betrachtet, da diese die Inputwerte sowieso nur weitergibt. Die Retina selbst und die statischen Gewichte dahinter werden also nicht weiter erwähnt oder abgebildet, da sie sowieso nicht informationsverarbeitend sind. Die Abbildung eines Perceptrons beginnt also bei den Inputneuronen.

SNIPE: Die Methoden `setSettingsTopologyFeedForward` sowie die Variante `-WithShortcuts` einer `NeuralNetworkDescriptor`-Instanz konfigurieren den Descriptor für FeedForward-Netze bzw. solche mit Shortcuts, indem sie die entsprechenden Verbindungsklassen erlauben. Andere Verbindungsklassen werden verboten und Fastprop wird aktiviert.

5.1 Das Singlelayerperceptron besitzt nur eine trainierbare Gewichtsschicht

Hier gehen von der Eingabeschicht Verbindungen mit trainierbaren Gewichten zu einem Ausgabeneuron Ω, welches ausgibt, ob das an den Eingabeneuronen eingegebene Muster erkannt wird oder nicht. Ein Singlelayerperceptron (kurz: SLP) besitzt also nur eine Ebene trainierbarer Gewichte (Abb. 5.1 auf der vorangehenden Seite).

Definition 5.4 (Singlelayerperceptron)**.** Als ***Singlelayerperceptron*** (***SLP***) wird ein Perceptron bezeichnet, welches nach der Schicht Eingabeneurone nur eine variable Gewichtsschicht und eine Schicht Ausgabeneurone Ω besitzt. Die technische Sicht eines SLPs findet sich in Abb. 5.2 auf der rechten Seite.

Es ändert am Prinzip des Perceptrons natürlich nichts wesentlich, wenn mehrere Ausgabeneurone $\Omega_1, \Omega_2, \ldots, \Omega_n$ vorkommen (Abb. 5.3 auf der rechten Seite): Ein Perceptron mit mehreren Ausgabeneuronen kann man auch als mehrere verschiedene Perceptrone mit derselben Eingabe ansehen.

Als triviale, zusammensetzbare Beispiele sind die in Abb. 5.4 auf Seite 94 dargestellten Booleschen Funktionen `AND` und `OR` zu sehen.

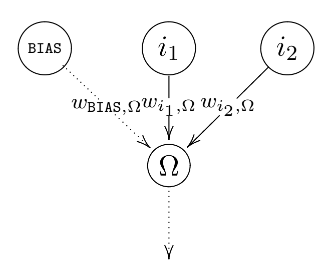


Abbildung 5.2: Ein Singlelayerperceptron mit zwei Eingabeneuronen und einem Outputneuron. Durch den aus dem Netz herausführenden Pfeil gibt das Netz die Ausgabe aus. In der Mitte befindet sich die trainierbare Schicht von Gewichten (beschriftet). Zur Erinnerung habe ich hier noch einmal das Biasneuron mit abgebildet. Obwohl das Gewicht $w_{\mathtt{BIAS},\Omega}$ ein ganz normales Gewicht ist und auch so behandelt wird, habe ich es hier nur gepunktet gezeichnet – dies erhöht die Übersichtlichkeit bei größeren Netzen stark. In Zukunft werden wir das Biasneuron nicht mehr mit abbilden.

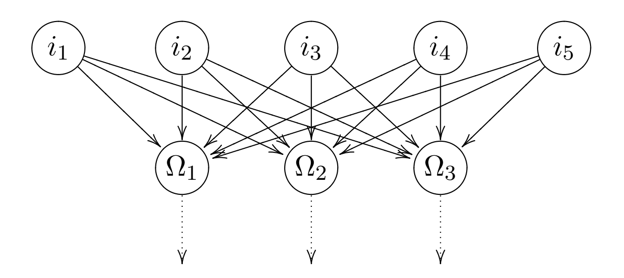


Abbildung 5.3: Singlelayerperceptron mit mehreren Ausgabeneuronen

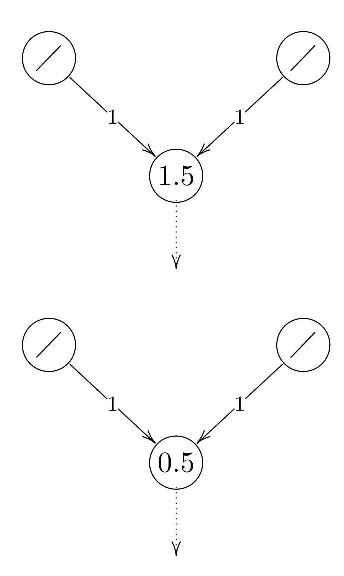


Abbildung 5.4: Zwei Singlelayerperceptrons für Boolesche Funktionen. Das obere Singlelayerperceptron realisiert ein AND, das untere ein OR. Die Aktivierungsfunktion des informationsverarbeitenden Neurons ist jeweils die binäre Schwellenwertfunktion, wo vorhanden stehen die Schwellenwerte in den Neuronen.

Wir möchten nun erfahren, wie wir ein Singlelayerperceptron trainieren können, und betrachten hierzu zunächst den Perceptron-Lernalgorithmus und anschließend die Delta-Regel.

5.1.1 Perceptron-Lernalgorithmus und Konvergenz-Theorem

Der ursprüngliche ***Perceptron-Lernalgorithmus*** mit binärer Aktivierungsfunktion in den Neuronen ist beschrieben in Alg. 1. Es ist bewiesen, dass der Algorithmus in endlicher Zeit konvergiert – das Perceptron also in endlicher Zeit alles lernen kann, was ihm möglich ist, zu repräsentieren (***Perceptron-Konvergenz-Theorem***, [Ros62]). Der Leser sollte sich hierbei nicht zu früh freuen. Was das Perceptron repräsentieren kann, werden wir noch erforschen.

Während der Erforschung der linearen Separierbarkeit von Problemen werden wir noch behandeln, dass zumindest das Singlelayerperceptron leider viele Probleme nicht repräsentieren kann.

5.1.2 Die Delta-Regel als gradientenbasiertes Lernverfahren für SLPs

Im Folgenden weichen wir von unserer binären Schwellenwertfunktion als Aktivierungsfunktion ab, denn zumindest für *Backpropagation of Error* brauchen wir, wie wir gleich sehen werden, eine *differenzierbare* oder gar *semilineare* Aktivierungsfunktion – für die nun folgende Delta-Regel (ebenfalls hergeleitet in [MR86]) ist sie nicht zwangsweise erforderlich, aber nützlich. Auf diesen Umstand wird aber auch noch einmal an Ort und Stelle hingewiesen. Die Delta-Regel hat gegenüber dem obigen Perceptron-Lernalgorithmus im Wesentlichen die Vorteile, für nicht-binäre Aktivierungsfunktionen geeignet zu sein und, bei großer Entfernung zum Lernziel, automatisch schneller zu lernen.

Angenommen, wir besitzen ein Singlelayerperceptron mit zufällig gesetzten Gewichten, dem wir eine Funktion anhand von Trainingsbeispielen beibringen möchten. Die Menge dieser Trainingsbeispiele nennen wir P – sie enthält, wie schon definiert, Paare (p, t) von Trainingsbeispielen p und zugehörigem Teaching Input t. Ich rufe auch noch einmal in Erinnerung, dass

- x Inputvektor und
- y Outputvektor eines Neuronalen Netzes ist,
- Outputneurone $\Omega_1, \Omega_2, \ldots, \Omega_{|O|}$ genannt werden und

```
 1: while ∃p ∈ P and Fehler zu groß do
 2:   Gebe ein p in Netz ein, berechne Ausgabe y {P Menge der Trainingsmuster}
 3:   for jedes Ausgabeneuron Ω do
 4:     if y_Ω = t_Ω then
 5:       Ausgabe richtig, keine Gewichtsänderung
 6:     else
 7:       if y_Ω = 0 then
 8:         for jedes Eingabeneuron i do
 9:           w_{i,Ω} := w_{i,Ω} + o_i {...Gewicht zu Ω um o_i vergrößern}
10:         end for
11:       end if
12:       if y_Ω = 1 then
13:         for jedes Eingabeneuron i do
14:           w_{i,Ω} := w_{i,Ω} − o_i {...Gewicht zu Ω um o_i verkleinern}
15:         end for
16:       end if
17:     end if
18:   end for
19: end while
```

Algorithmus 1: Perceptron-Lernalgorithmus. Der Perceptron-Lernalgorithmus verringert Gewichte zu Ausgabeneuronen, welche 1 statt 0 ausgeben, und erhöht Gewichte im umgekehrten Fall.

- ▷ i Input sowie
- ▷ o Output eines Neurons ist.

Ferner haben wir definiert, dass

- ▷ der Fehlervektor E_p die Differenz $(t - y)$ unter einem bestimmten Trainingsbeispiel p darstellt.
- ▷ Sei weiterhin wie gehabt O die Menge der Ausgabeneurone und
- ▷ I die Menge der Eingabeneurone.

Als weitere Namenskonvention wollen wir vereinbaren, dass z.B. für Output o und Teaching Input t ein zusätzlicher Index p gesetzt werden darf, um anzuzeigen, dass diese Größe musterspezifisch ist – dies erhöht manchmal die Übersichtlichkeit ganz erheblich.

Unser Lernziel ist jetzt natürlich, dass bei allen Trainingsbeispielen der Output y des Netzes annähernd gleich dem gewünschten Output t ist, also formal gilt

$$\forall p : y \approx t \quad \text{bzw.} \quad \forall p : E_p \approx 0.$$

Hierfür müssen wir erst lernen, den Gesamtfehler Err als Funktion der Gewichte zu betrachten: Der Gesamtfehler nimmt zu oder ab, je nachdem, wie wir die Gewichte ändern.

Definition 5.5 (Fehlerfunktion)**.** Die ***Fehlerfunktion***

$$\text{Err} : W \to \mathbb{R}$$

fasst die Menge[2] der Gewichte W als Vektor auf und bildet die Gewichtswerte auf den normalisierten Ausgabefehler ab (normalisiert daher, weil man sämtliche Ausgabefehler sonst nicht in einem einzelnen $e \in \mathbb{R}$ abbilden kann, um einen Gradientenabstieg darauf durchzuführen). Dass sich analog eine ***spezifische Fehlerfunktion*** für ein einzelnes Muster p bilden lässt, ist offensichtlich.

Wie wir bereits in Abschnitt 4.5 zum Thema Gradientenabstiegsverfahren gesehen haben, berechnen Gradientenabstiegsverfahren den Gradienten einer beliebig- aber endlichdimensionalen Funktion (hier der Fehlerfunktion $\text{Err}(W)$) und gehen entgegen dem Gradienten nach unten, bis ein Minimum erreicht ist. $\text{Err}(W)$ ist auf der Menge sämtlicher Gewichte definiert, die wir hier als Vektor W ansehen. Es wird also versucht,

2 Der Tradition anderer Literatur folgend, habe ich W vorher als Gewichts*matrix* definiert – ich bin mir dieses Konfliktes bewusst, er wird uns hier aber nicht weiter stören.

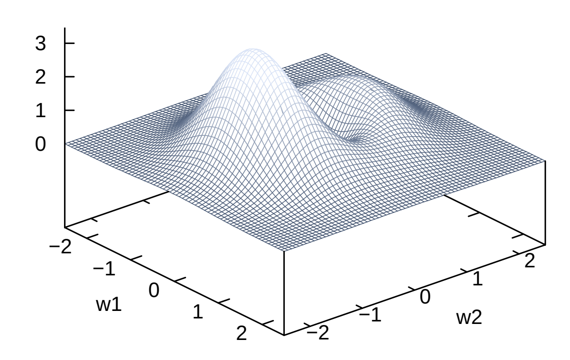


Abbildung 5.5: Beispielhafte Fehlerfläche eines Neuronalen Netzes mit zwei trainierbaren Verbindungen w_1 und w_2. In der Regel haben Neuronale Netze mehr als zwei Verbindungen, was hier nicht so übersichtlich abzubilden gewesen wäre. Weiterhin ist die Fehlerfläche meist sehr viel zerklüfteter, was die Suche nach dem Minimum erschwert.

den Fehler zu verringern bzw. zu minimieren, indem man, salopp gesagt, an den Gewichten dreht – man bekommt also Informationen darüber, wie man die Gewichte verändern soll (die Veränderung aller Gewichte bezeichnen wir mit ΔW), indem man die Fehlerfunktion $\mathrm{Err}(W)$ nach diesen ableitet:

$$\Delta W \sim -\nabla \mathrm{Err}(W). \tag{5.1}$$

Aufgrund dieser Proportionalität gibt es eine Proportionalitätskonstante η, so dass Gleichheit gilt (η wird bald noch eine weitere Bedeutung und wirklich praktischen Nutzen außerhalb der bloßen Bedeutung als Proportionalitätskonstante bekommen. Ich möchte den Leser bitten, sich bis dahin noch etwas zu gedulden.):

$$\Delta W = -\eta \nabla \mathrm{Err}(W). \tag{5.2}$$

Die Ableitung der Fehlerfunktion nach den Gewichten schreiben wir jetzt als normale partielle Ableitung nach einem Gewicht $w_{i,\Omega}$ (es gibt nur variable Gewichte zu Ausgabeneuronen Ω), um damit rechnerisch etwas mehr anfangen zu können. Wir drehen also an jedem einzelnen Gewicht und schauen, wie sich die Fehlerfunktion dabei ändert, leiten also die Fehlerfunktion nach einem Gewicht $w_{i,\Omega}$ ab und erhalten so die Information $\Delta w_{i,\Omega}$, wie wir dieses Gewicht verändern sollen.

$$\Delta w_{i,\Omega} = -\eta \frac{\partial \mathrm{Err}(W)}{\partial w_{i,\Omega}}. \tag{5.3}$$

Nun stellt sich langsam die Frage: Wie ist denn genau unsere Fehlerfunktion definiert? Es ist schlecht für uns, wenn sich viele Ergebnisse fern der gewünschten finden, die

Fehlerfunktion sollte dann also große Werte liefern – auf der anderen Seite ist es auch nicht gut, wenn viele Ergebnisse nahe der gewünschten sind, es aber vielleicht einen *sehr* weit entfernten Ausreißer gibt. Es bietet sich also der ***Quadratische Abstand*** zwischen dem Ausgabevektor y und dem Teaching Input t an, der uns den für ein Trainingsbeispiel p spezifischen Fehler Err_p über die Ausgabe aller Outputneurone Ω liefert:

$$\text{Err}_p(W) = \frac{1}{2} \sum_{\Omega \in O} (t_{p,\Omega} - y_{p,\Omega})^2. \tag{5.4}$$

Wir quadrieren also die Differenzen der Komponenten der Vektoren t und y unter einem Muster p und summieren diese Quadrate auf. Die Fehlerdefinition Err und damit die der Fehlerfunktion $\text{Err}(W)$ ergibt sich dann einfach durch die Aufsummierung der spezifischen Fehler $\text{Err}_p(W)$ aller Muster p:

$$\text{Err}(W) = \sum_{p \in P} \text{Err}_p(W) \tag{5.5}$$

$$= \frac{1}{2} \overbrace{\sum_{p \in P} \underbrace{\left(\sum_{\Omega \in O} (t_{p,\Omega} - y_{p,\Omega})^2 \right)}_{\text{Summe über alle } \Omega}}^{\text{Summe über alle } p}. \tag{5.6}$$

Der aufmerksame Leser wird sich natürlich fragen, woher denn in Gleichung 5.4 der Faktor $\frac{1}{2}$ plötzlich kommt, und wo denn, da die Gleichung dem euklidischen Abstand so ähnlich sieht, die Wurzel geblieben ist. Beides folgt aus einfacher Pragmatik: Es geht nur um die Fehlerminimierung. Die Wurzelfunktion ist monoton und sinkt mit ihrem Argument, also können wir sie auch dem Rechen- und Implementationsaufwand zuliebe weglassen, da wir sie für die Minimierung nicht brauchen. Ebenso ist egal, ob wir den zu minimierenden Term durch den Vorfaktor $\frac{1}{2}$ halbieren: Ich darf also mit $\frac{1}{2}$ multiplizieren – aus der reinen Faulheit heraus, damit es sich im weiteren Verlauf unserer Rechnungen gegen eine 2 herauskürzt.

Nun wollen wir fortfahren, die Delta-Regel für lineare Aktivierungsfunktionen herzuleiten. Wir haben bereits behandelt, dass man etwas an den einzelnen Gewichten $w_{i,\Omega}$ dreht und schaut, wie sich der Fehler $\text{Err}(W)$ verändert – was der Ableitung der Fehlerfunktion $\text{Err}(W)$ nach eben diesem Gewicht $w_{i,\Omega}$ entspricht. Diese Ableitung entspricht

(da sich der Gesamtfehler $\mathrm{Err}(W)$ aus der Summe der spezifischen Fehler ergibt) der Summe der Ableitungen aller spezifischen Fehler Err_p nach diesem Gewicht:

$$\Delta w_{i,\Omega} = -\eta \frac{\partial \mathrm{Err}(W)}{\partial w_{i,\Omega}} \tag{5.7}$$

$$= \sum_{p \in P} -\eta \frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}}. \tag{5.8}$$

An dieser Stelle möchte ich noch einmal darüber nachdenken, wie ein Neuronales Netz denn Daten verarbeitet. Im Grunde werden die Daten nur durch eine Funktion geschickt, das Ergebnis der Funktion durch eine weitere und so weiter und so fort. Lassen wir die Outputfunktion einmal außen vor, so besteht der Weg von Neuronenausgaben o_{i_1} und o_{i_2}, die von Neuronen i_1 und i_2 in ein Neuron Ω eingegeben werden, zunächst aus der Propagierungsfunktion (hier gewichtete Summe), aus der wir dann die Netzeingabe erhalten. Diese wird dann durch die Aktivierungsfunktion des Neurons Ω geschickt, so dass wir den Output dieses Neurons erhalten, der auch gleichzeitig eine Komponente des Ausgabevektors y ist:

$$\begin{aligned} \mathrm{net}_\Omega &\rightarrow f_{\mathrm{act}} \\ &= f_{\mathrm{act}}(\mathrm{net}_\Omega) \\ &= o_\Omega \\ &= y_\Omega. \end{aligned}$$

Wie wir sehen, resultiert dieser Output aus vielen ineinander geschachtelten Funktionen:

$$o_\Omega = f_{\mathrm{act}}(\mathrm{net}_\Omega) \tag{5.9}$$

$$= f_{\mathrm{act}}(o_{i_1} \cdot w_{i_1,\Omega} + o_{i_2} \cdot w_{i_2,\Omega}). \tag{5.10}$$

Dass wir den Output auch bis in die Eingabeneurone aufschlüsseln können, ist klar (das ist hier nicht notwendig, da diese bei einem SLP keine Informationsverarbeitung betreiben). Wir wollen also die Ableitungen von Gleichung 5.8 durchführen und können durch die Funktionsschachtelung die *Kettenregel* anwenden, um die in Gleichung 5.8 enthaltene Ableitung $\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}}$ zu zerlegen.

$$\frac{\partial \mathrm{Err}_p(W)}{\partial w_{i,\Omega}} = \frac{\partial \mathrm{Err}_p(W)}{\partial o_{p,\Omega}} \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}}. \tag{5.11}$$

Betrachten wir den ersten multiplikativen Faktor der obigen Gleichung 5.11 auf der linken Seite, der die Ableitung des spezifischen Fehlers $\text{Err}_p(W)$ nach dem Output darstellt, also die Veränderung des Fehlers Err_p mit dem Output $o_{p,\Omega}$: Es ist bei Betrachtung von Err_p (Gleichung 5.4 auf Seite 99) klar, dass diese Veränderung sich genau mit der Differenz zwischen Teaching Input und Ausgabe $(t_{p,\Omega} - o_{p,\Omega})$ verändert (wir erinnern uns: Da Ω Ausgabeneuron, gilt $o_{p,\Omega} = y_{p,\Omega}$). Ist der Output dem Teaching Input näher, so ist der spezifische Fehler kleiner. Wir können also das eine durch das andere ersetzen, wobei wir diese Differenz auch $\delta_{p,\Omega}$ nennen (daher hat die Delta-Regel ihren Namen):

$$\frac{\partial \text{Err}_p(W)}{\partial w_{i,\Omega}} = -(t_{p,\Omega} - o_{p,\Omega}) \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}} \tag{5.12}$$

$$= -\delta_{p,\Omega} \cdot \frac{\partial o_{p,\Omega}}{\partial w_{i,\Omega}} \tag{5.13}$$

Der zweite multiplikative Faktor der Gleichung 5.11 auf der linken Seite und der folgenden ist die Ableitung des Outputs des Neurons Ω zum Muster p nach dem Gewicht $w_{i,\Omega}$. Wie verändert sich also $o_{p,\Omega}$ bei der Veränderung des Gewichts von i nach Ω? Da wir nach der Forderung am Anfang der Herleitung nur eine lineare Aktivierungsfunktion f_{act} haben, können wir genauso gut die Veränderung der Netzeingabe bei Veränderung von $w_{i,\Omega}$ betrachten:

$$\frac{\partial \text{Err}_p(W)}{\partial w_{i,\Omega}} = -\delta_{p,\Omega} \cdot \frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}}. \tag{5.14}$$

Diese Ableitung $\frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}}$ können wir nun vereinfachen: Die abzuleitende Funktion $\sum_{i \in I}(o_{p,i} w_{i,\Omega})$ besteht aus vielen Summanden, und nur der Summand $o_{p,i} w_{i,\Omega}$ enthält die Variable $w_{i,\Omega}$, nach der wir ableiten. Es gilt also $\frac{\partial \sum_{i \in I}(o_{p,i} w_{i,\Omega})}{\partial w_{i,\Omega}} = o_{p,i}$ und damit:

$$\frac{\partial \text{Err}_p(W)}{\partial w_{i,\Omega}} = -\delta_{p,\Omega} \cdot o_{p,i} \tag{5.15}$$

$$= -o_{p,i} \cdot \delta_{p,\Omega}. \tag{5.16}$$

Dies setzen wir in die Gleichung 5.8 auf der linken Seite ein und erhalten so unsere Modifikationsregel für ein Gewicht $w_{i,\Omega}$:

$$\Delta w_{i,\Omega} = \eta \cdot \sum_{p \in P} o_{p,i} \cdot \delta_{p,\Omega}. \tag{5.17}$$

Allerdings: Wir haben die Herleitung schon von Anfang an als Offline-Regel begonnen, indem wir uns Gedanken gemacht haben, wie wir die Fehler aller Muster aufsummieren und jeweils *nach* der Präsentation aller Muster lernen. Dies ist der mathematisch korrekte Weg, aber aufwändiger zu implementieren und, wie wir später in diesem Kapitel sehen werden, auch teilweise rechenaufwändiger während des Trainings.

Für die „Online-Learning-Version" der Delta-Regel wird die Aufsummierung einfach weggelassen und direkt nach der Präsentation jedes Musters gelernt, was uns auch die Schreibweise vereinfacht (sie muss nicht mehr auf ein Muster p bezogen sein):

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot \delta_\Omega. \tag{5.18}$$

Diese Version der Delta-Regel möchte ich auch für die folgende Definition verwenden.

Definition 5.6 (Delta-Regel)**.** Bestimmen wir analog zu obiger Herleitung, dass die Funktion h aus der Hebb-Regel (Gleichung 4.6 auf Seite 86) nur den Ausgabewert o_i des Vorgängerneurons i wieder ausgibt und die Funktion g die Differenz von gewünschter Aktivierung t_Ω und tatsächlicher Aktivierung a_Ω ist, so erhalten wir die ***Delta-Regel***, auch bekannt als ***Widrow-Hoff-Regel***:

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - a_\Omega) = \eta o_i \delta_\Omega \tag{5.19}$$

Wenn man als Teaching Input die gewünschte Ausgabe anstatt Aktivierung anlegt, die Ausgabefunktion der Outputneurone also keine Identität darstellt, erhält man

$$\Delta w_{i,\Omega} = \eta \cdot o_i \cdot (t_\Omega - o_\Omega) = \eta o_i \delta_\Omega \tag{5.20}$$

und δ_Ω entspricht dann der Differenz zwischen t_Ω und o_Ω.

Bei der Delta-Regel ist die Gewichtsänderung aller Gewichte zu einem Ausgabeneuron Ω proportional

- ▷ zur Differenz der aktuellen Aktivierung bzw. Ausgabe a_Ω bzw. o_Ω und dem dazugehörigen Teaching Input t_Ω. Diesen Faktor möchten wir δ_Ω nennen, er wird auch „***Delta***" gesprochen.

Offensichtlich gilt die Delta-Regel jedoch nur für SLPs, da sich die Formel immer auf den Teaching Input bezieht und für innere Verarbeitungsschichten von Neuronen *kein Teaching Input existiert.*

Ein. 1	Ein. 2	Ausgabe
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 5.1: Definition des logischen XORs. Links die Eingabewerte, rechts die definierte Ausgabe.

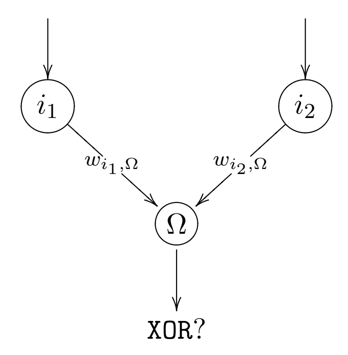


Abbildung 5.6: Skizze für ein Singlelayerperceptron, welches die XOR-Funktion darstellen soll – solch ein SLP kann es aber nicht geben.

5.2 Ein SLP kann nur linear separierbare Daten repräsentieren

Sei f die XOR-Funktion, welche zwei binäre Eingaben erwartet und eine binäre Ausgabe erzeugt (genaue Definition siehe Tabelle 5.1).

Versuchen wir, durch ein SLP mit zwei Eingabeneuronen i_1, i_2 und einem Ausgabeneuron Ω die XOR-Funktion darzustellen (Abb. 5.6).

Wir verwenden hier die gewichtete Summe als Propagierungsfunktion, eine binäre Aktivierungsfunktion mit Schwellenwert Θ und die Identität als Ausgabefunktion. Ω muss also in Abhängigkeit von i_1 und i_2 den Wert 1 ausgeben, wenn gilt:

$$\text{net}_\Omega = o_{i_1} w_{i_1,\Omega} + o_{i_2} w_{i_2,\Omega} \geq \Theta_\Omega \tag{5.21}$$

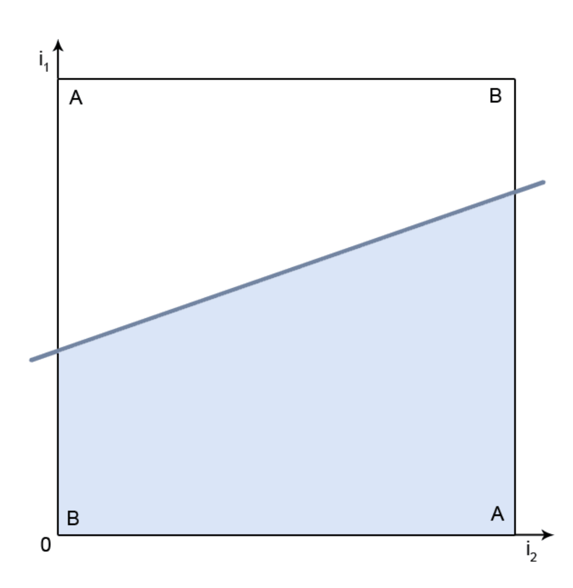


Abbildung 5.7: Lineare Separierung von $n = 2$ Eingaben von Inputneuronen i_1 und i_2 durch 1-dimensionale Gerade. A und B bezeichnen die Zugehörigkeit der Eckpunkte zu den zu separierenden Mengen der XOR-Funktion.

Gehen wir von einem positiven Gewicht $w_{i_2,\Omega}$ aus, so ist Ungleichung 5.21 auf der vorangehenden Seite äquivalent zu der Ungleichung

$$o_{i_1} \geq \frac{1}{w_{i_1,\Omega}}(\Theta_\Omega - o_{i_2} w_{i_2,\Omega}) \tag{5.22}$$

Bei konstantem Schwellenwert Θ_Ω stellt der rechte Teil der Ungleichung 5.22 eine Gerade durch ein von den möglichen Ausgaben o_{i_1} und o_{i_2} der Eingabeneurone i_1 und i_2 aufgespanntes Koordinatensystem (Abb. 5.7) dar.

Für ein (wie für Ungleichung 5.22 gefordertes) positives $w_{i_2,\Omega}$ feuert das Ausgabeneuron Ω bei den Eingabekombinationen, welche *über* der erzeugten Geraden liegen. Für ein negatives $w_{i_2,\Omega}$ würde es für alle Eingabekombinationen feuern, welche *unter* der Geraden liegen. Es sei angemerkt, dass nur die vier Eckpunkte des Einheitsquadrates mögliche Eingaben sind, da die XOR-Funktion nur binäre Eingaben kennt.

Um das XOR-Problem zu lösen, müssen wir also die Gerade so drehen und verschieben, dass sie die Eingabemenge $A = \{(0,0),(1,1)\}$ von der Eingabemenge $B = \{(0,1),(1,0)\}$ abgrenzt - was offensichtlich nicht möglich ist.

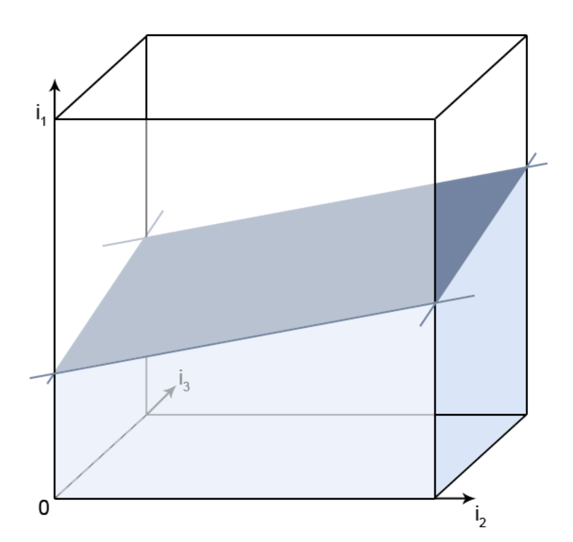


Abbildung 5.8: Lineare Separierung von $n = 3$ Eingaben von Inputneuronen i_1, i_2 und i_3 durch 2-dimensionale Ebene.

Allgemein lassen sich die Eingabemöglichkeiten n vieler Eingabeneurone in einem n-dimensionalen Würfel darstellen, der von einem SLP durch eine $(n-1)$-dimensionale Hyperebene separiert wird (Abb. 5.8) – nur Mengen, die durch eine solche Hyperebene trennbar, also ***linear separierbar*** sind, kann ein SLP klassifizieren.

Es spricht leider viel dafür, dass der Prozentsatz der linear separierbaren Probleme mit steigendem n schnell abnimmt (siehe Tabelle 5.2 auf der folgenden Seite), was die Funktionalität des SLPs einschränkt – weiterhin sind Prüfungen auf lineare Separierbarkeit schwierig. Für schwierigere Aufgaben mit mehr Eingaben benötigen wir also etwas Mächtigeres als das SLP. Das XOR-Problem stellt schon eine dieser Aufgaben dar, braucht doch ein Perceptron, das die XOR-Funktion repräsentieren will, bereits eine verdeckte Ebene (Abb. 5.9 auf der folgenden Seite).

5.3 Ein Multilayerperceptron enthält mehr trainierbare Gewichtsschichten

Mächtiger als ein SLP ist ein Perceptron mit zwei oder mehr trainierbaren Gewichtsschichten (Multilayerperceptron bzw. MLP genannt). Wie wir wissen, kann ein Single-

n	Anzahl binärer Funktionen	davon lin. separier-bar	Anteil
1	4	4	100%
2	16	14	87.5%
3	256	104	40.6%
4	$65,536$	$1,772$	2.7%
5	$4.3 \cdot 10^9$	$94,572$	0.002%
6	$1.8 \cdot 10^{19}$	$5,028,134$	$\approx 0\%$

Tabelle 5.2: Anzahl der Funktionen bezüglich n binärer Eingaben und Anzahl und Anteil der davon linear separierbaren Funktionen. Nach [Zel94, Wid89, Was89].

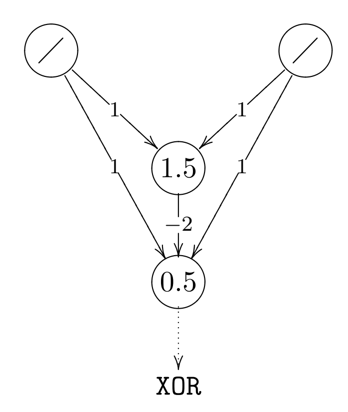


Abbildung 5.9: Neuronales Netz, welches die XOR-Funktion realisiert. Schwellenwerte stehen (soweit vorhanden) innerhalb der Neurone.

layerperceptron den Inputraum durch eine Hyperebene (bei zweidimensionalem Inputraum eine Geraden) teilen. Ein zweistufiges Perceptron (2 trainierbare Gewichtsschichten, 3 Schichten Neurone) kann *konvexe Polygone* klassifizieren, indem es diese Geraden weiterbehandelt, zum Beispiel in der Form „erkenne Muster, die über Gerade 1, unter Gerade 2 und unter Gerade 3 liegen". Wir haben also bildlich gesprochen ein SLP mit mehreren Ausgabeneuronen genommen und ein weiteres SLP „angehangen" (oberer Teil der Abb. 5.10 auf der folgenden Seite). Ein Multilayerperceptron stellt einen ***universellen Funktionsapproximator*** dar, wie aus dem *Cybenko-Theorem* hervorgeht [Cyb89].

Eine weitere trainierbare Gewichtsschicht verfährt genauso, nur eben mit den konvexen Polygonen, die nun wieder durch eine Gewichtsschicht aufeinander addiert, voneinander abgezogen oder mit anderen Operationen nachbearbeitet werden können (unterer Teil der Abb. 5.10 auf der folgenden Seite).

Allgemein kann mathematisch bewiesen werden, dass schon ein Multilayerperceptron mit einer Schicht versteckter Neurone eine Funktion mit endlich vielen Unstetigkeitsstellen sowie deren erste Ableitung beliebig genau approximieren kann – leider ist der Beweis aber nicht konstruktiv, und so ist es uns selbst überlassen, die richtige Neuronenanzahl und Gewichte zu finden.

Im Folgenden möchten wir für verschiedene Multilayerperceptrons eine Kurzschreibweise verwenden, welche weite Verbreitung findet: So ist ein zweistufiges Perceptron mit 5 Neuronen in der Eingabeschicht, 3 Neuronen in der versteckten Schicht und 4 Neuronen in der Ausgabeschicht ein 5-3-4-MLP.

Definition 5.7 (Multilayerperceptron)**.** Perceptrons mit mehr als einer Schicht variabel gewichteter Verbindungen bezeichnen wir als ***Multilayerperceptron*** (***MLP***). Ein n-Layer-Perceptron bzw. n-stufiges Perceptron hat dabei genau n variable Gewichtsschichten und $n+1$ Schichten Neurone (die Retina lassen wir außer Acht), die Neuronenschicht 1 ist hierbei die Schicht Eingabeneurone.

Da dreistufige Perceptrons durch Vereinigung und Schnitt beliebig vieler konvexer Polygone Mengen beliebiger Form klassifizieren können, bringt uns eine weitere Stufe für Funktionsdarstellungen durch Neuronale Netze keinen Vorteil mehr. Vorsicht beim Literaturlesen: An der Schichtendefinition scheiden sich die Geister. Manche Quellen zählen die Neuronenschichten, manche die Gewichtsschichten. Manche zählen die Retina hinzu, manche zählen die trainierbaren Gewichtsschichten. Manche zählen (aus welchen Gründen auch immer) die Ausgabeneuronenschicht nicht mit. Ich habe hier die Definition gewählt, die meiner Meinung nach am meisten über die Lernfähigkeiten aussagt – und werde sie hier auch konsistent durchhalten. Nochmal zur Erinnerung:

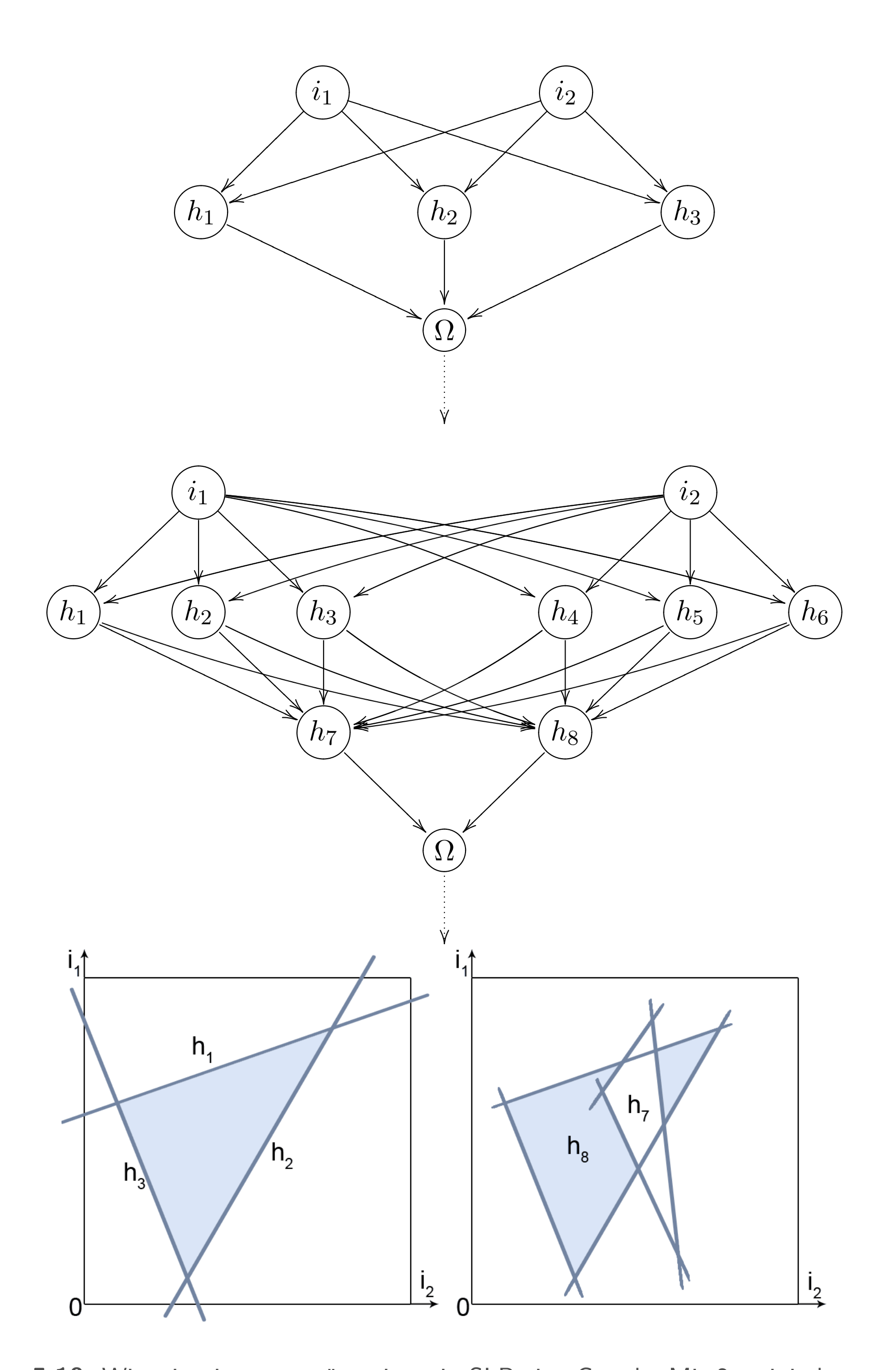


Abbildung 5.10: Wie wir wissen, repräsentiert ein SLP eine Gerade. Mit 2 trainierbaren Gewichtsschichten kann man mehrere Geraden zu konvexen Polygonen zusammensetzen (oben). Unter Verwendung von 3 trainierbaren Gewichtsschichten kann man mit mehreren Polygonen beliebige Mengen modellieren (unten).

n	**klassifizierbare Menge**
1	Hyperebene
2	konvexes Polygon
3	jede beliebige Menge
4	auch jede beliebige Menge, also kein weiterer Vorteil

Tabelle 5.3: Hier wird dargestellt, mit welchem Perceptron sich Mengen welcher Art klassifizieren lassen, wobei das n die Anzahl der trainierbaren Gewichtsschichten darstellt.

Ein n-stufiges Perceptron besitzt genau n trainierbare Gewichtsschichten. Eine Zusammenfassung, welche Perceptrons welche Art von Menge klassifizieren können, findet sich noch einmal in Tabelle 5.3. Wir werden uns jetzt der Herausforderung widmen, Perceptrons mit mehr als einer Gewichtsschicht zu trainieren.

5.4 Backpropagation of Error verallgemeinert die Delta-Regel auf MLPs

Im Folgenden möchte ich die ***Backpropagation of Error***-Lernregel (Kurz: Backpropagation, Backprop oder auch BP) herleiten und näher erläutern, mit der man mehrstufige Perceptrons, welche *semilineare*[3] Aktivierungsfunktionen besitzen, trainieren kann. Binäre Schwellenwertfunktionen und sonstige nicht-differenzierbare Funktionen werden *nicht* mehr unterstützt, das macht aber nichts: Wir haben ja gesehen, dass man die Fermifunktion bzw. den Tangens Hyperbolicus durch einen Temperatur-Parameter T der binären Schwellenwertfunktion beliebig annähern kann. Weitgehend werde ich der Herleitung nach [Zel94] bzw. [MR86] folgen – ich möchte aber noch einmal darauf hinweisen, dass das Verfahren bereits früher von Paul Werbos in [Wer74] publiziert wurde, jedoch wesentlich weniger Leser fand als in [MR86].

Backpropagation ist ein Gradientenabstiegsverfahren (mit all den Stärken und Schwächen des Gradientenabstiegs), wobei die Fehlerfunktion $\text{Err}(W)$ hier sämtliche n Gewichte als Argument entgegennimmt (Abb. 5.5 auf Seite 98) und diese dem Ausgabefehler zuordnet, also n-dimensional ist. Auf $\text{Err}(W)$ sucht man durch Gradientenabstieg einen Punkt geringen oder gar geringsten Fehlers. Backpropagation trainiert also wie die Delta-Regel die Gewichte des Neuronalen Netzes – und genau die Delta-Regel bzw.

3 Semilineare Funktionen sind monoton und differenzierbar – aber im Allgemeinen nicht linear.

ihre Größe δ_i für ein Neuron i wird durch Backpropagation von einer auf mehrere trainierbare Gewichtsschichten *erweitert*.

5.4.1 Die Herleitung erfolgt völlig analog zur Deltaregel, aber mit allgemeinerem Delta

Im Vorfeld sei definiert, dass sich die Netzeingabe der einzelnen Neurone i durch die gewichtete Summe ergibt. Weiterhin seien $o_{p,i}$, $\text{net}_{p,i}$ etc. wie schon bei Herleitung der Delta-Regel definiert als die gewohnten o_i, net_i, etc. unter dem Eingabemuster p, welches wir zum Trainieren verwenden. Auch sei die Ausgabefunktion wieder die Identität, es gilt also $o_i = f_{\text{act}}(\text{net}_{p,i})$ für jedes Neuron i. Da es sich um eine Verallgemeinerung der Delta-Regel handelt, benutzen wir wieder das gleiche Formelgerüst wie bei der Delta-Regel (Gleichung 5.20 auf Seite 102). Was wir verallgemeinern müssen, ist, wie schon angedeutet, die Größe δ für jedes Neuron.

Zunächst: Wo befindet sich das Neuron, für das wir ein δ errechnen wollen? Es liegt nahe, ein beliebiges inneres Neuron h zu wählen, welches eine Menge K von Vorgängerneuronen k sowie eine Menge L von Nachfolgerneuronen l besitzt, welche *ebenfalls innere Neurone* sind (Siehe Abb. 5.11 auf der rechten Seite). Es ist dabei irrelevant, ob die Vorgängerneurone bereits die Eingabeneurone sind.

Wir führen nun das gleiche Spiel wie bei der Herleitung der Delta-Regel durch und spalten Funktionen durch die Kettenregel. Ich werde bei dieser Herleitung nicht ganz so ausführlich sein, das Prinzip ist aber dem der Delta-Regel ähnlich (die Unterschiede liegen eben wie gesagt im verallgemeinerten δ). Wir leiten also zunächst die Fehlerfunktion Err nach einem Gewicht $w_{k,h}$ ab.

$$\frac{\partial \text{Err}(w_{k,h})}{\partial w_{k,h}} = \underbrace{\frac{\partial \text{Err}}{\partial \text{net}_h}}_{=-\delta_h} \cdot \frac{\partial \text{net}_h}{\partial w_{k,h}} \tag{5.23}$$

Der erste Faktor der Gleichung 5.23 ist $-\delta_h$, welches wir gleich noch betrachten wollen. Der zweite Faktor der Gleichung trägt im Zähler die Netzeingabe, also die gewichtete Summe, so dass wir diese auch direkt ableiten können. Es fallen wieder alle Summanden der Summe weg bis auf denjenigen, der $w_{k,h}$ enthält. Dieser Summand heißt $w_{k,h} \cdot o_k$. Leitet man diesen ab, bleibt also der Output des Neurons k übrig:

$$\frac{\partial \text{net}_h}{\partial w_{k,h}} = \frac{\partial \sum_{k \in K} w_{k,h} o_k}{\partial w_{k,h}} \tag{5.24}$$

$$= \boxed{o_k} \tag{5.25}$$

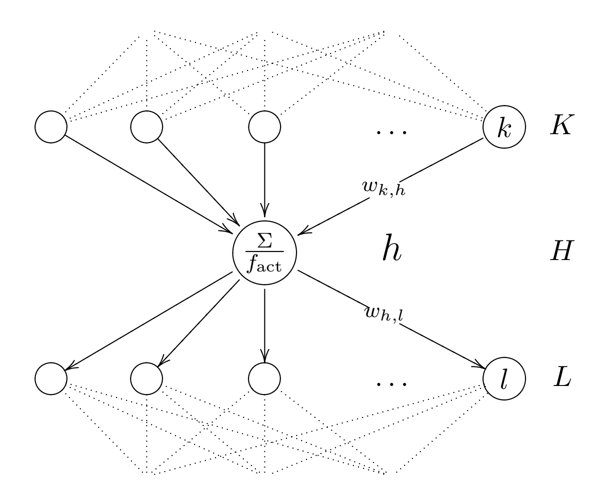


Abbildung 5.11: Skizze der Lage unseres Neurons h im Neuronalen Netz. Es liegt in der Schicht H, Vorgängerschicht ist K, nachfolgende Schicht ist L.

Wie versprochen, behandeln wir nun das $-\delta_h$ der Gleichung 5.23 auf der linken Seite, welches wir wieder mit der Kettenregel aufspalten:

$$\delta_h = -\frac{\partial \text{Err}}{\partial \text{net}_h} \tag{5.26}$$

$$= -\frac{\partial \text{Err}}{\partial o_h} \cdot \frac{\partial o_h}{\partial \text{net}_h} \tag{5.27}$$

Die Ableitung des Outputs nach der Netzeingabe (der zweite Faktor in Gleichung 5.27) kommt natürlich der Ableitung der Aktivierungsfunktion nach der Netzeingabe gleich:

$$\frac{\partial o_h}{\partial \text{net}_h} = \frac{\partial f_{\text{act}}(\text{net}_h)}{\partial \text{net}_h} \tag{5.28}$$

$$= \boxed{{f_{\text{act}}}'(\text{net}_h)} \tag{5.29}$$

Analog leiten wir nun den ersten Faktor in Gleichung 5.27 ab. Diese Stelle möge sich der Leser bitte gut durch den Kopf gehen lassen. Wir müssen uns hierfür nur klarmachen, dass *die Ableitung der Fehlerfunktion nach dem Output einer inneren Neuronenschicht*

abhängig ist vom Vektor sämtlicher Netzeingaben der Nachfolgeschicht. Dies schlägt in Gleichung 5.30 zu Buche:

$$-\frac{\partial \mathrm{Err}}{\partial o_h} = -\frac{\partial \mathrm{Err}(\mathrm{net}_{l_1}, \dots, \mathrm{net}_{l_{|L|}})}{\partial o_h} \tag{5.30}$$

Nach der Definition der mehrdimensionalen Kettenregel folgt dann sofort die Gleichung 5.31:

$$-\frac{\partial \mathrm{Err}}{\partial o_h} = \sum_{l \in L} \left(-\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_l} \cdot \frac{\partial \mathrm{net}_l}{\partial o_h} \right) \tag{5.31}$$

Die Summe in Gleichung 5.31 besitzt zwei Faktoren. Mit diesen Faktoren, summiert über die Nachfolgeschicht L, wollen wir uns nun beschäftigen. Wir rechnen den zweiten Faktor in der folgenden Gleichung 5.33 einfach aus:

$$\frac{\partial \mathrm{net}_l}{\partial o_h} = \frac{\partial \sum_{h \in H} w_{h,l} \cdot o_h}{\partial o_h} \tag{5.32}$$

$$= \boxed{w_{h,l}} \tag{5.33}$$

Analog gilt für den ersten Faktor nach der Definition unseres δ:

$$-\frac{\partial \mathrm{Err}}{\partial \mathrm{net}_l} = \boxed{\delta_l} \tag{5.34}$$

Wir setzen nun ein:

$$\Rightarrow -\frac{\partial \mathrm{Err}}{\partial o_h} = \sum_{l \in L} \delta_l w_{h,l} \tag{5.35}$$

Eine graphisch aufbereitete Version der δ-Verallgemeinerung mit allen Aufspaltungen findet sich in Abbildung 5.12 auf der rechten Seite.

Dem Leser wird bereits aufgefallen sein, dass einige Zwischenergebnisse umrahmt wurden. Umrahmt wurden genau die Zwischenergebnisse, die in der Gewichtsveränderung von $w_{k,h}$ einen Faktor ausmachen. Führt man die obigen Gleichungen mit den umrahmten Zwischenergebnissen zusammen, so ergibt sich die gesuchte Gewichtsänderung $\Delta w_{k,h}$ zu

$$\Delta w_{k,h} = \eta o_k \delta_h \text{ mit} \tag{5.36}$$

$$\delta_h = f'_{\mathrm{act}}(\mathrm{net}_h) \cdot \sum_{l \in L} (\delta_l w_{h,l})$$

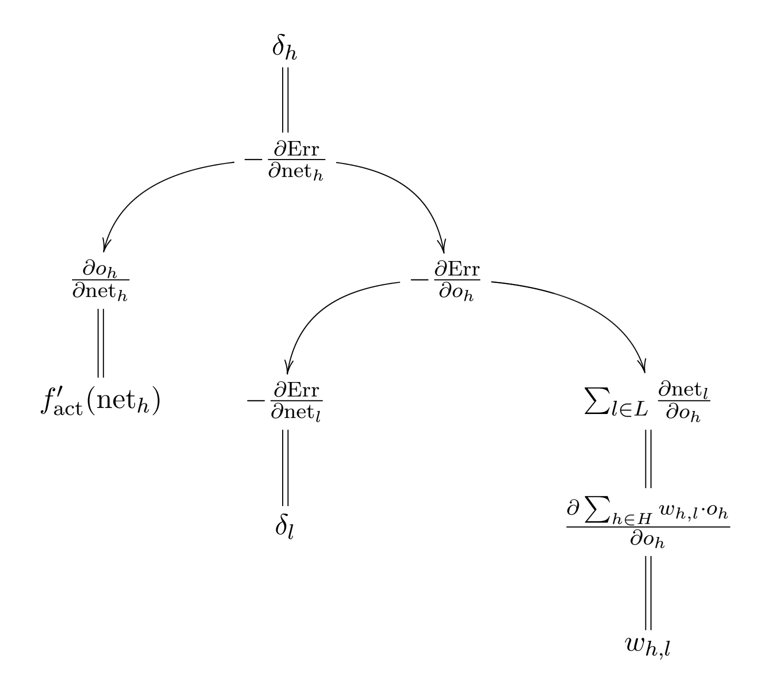


Abbildung 5.12: Graphische Darstellung der Gleichsetzungen (durch langgezogene Gleichzeichen) und Kettenregel-Aufspaltungen (durch Pfeile) im Rahmen der Herleitung von Backpropagation. Die Blätter des Baumes spiegeln die in der Herleitung umrahmten Endergebnisse aus der Verallgemeinerung des δ wieder.

– natürlich nur für den Fall, dass h ein inneres Neuron ist (sonst gibt es ja auch keine Nachfolgeschicht L).

Den Fall, dass h Ausgabeneuron ist, haben wir ja schon in der Herleitung der Delta-Regel behandelt. Insgesamt ergibt sich also unsere Verallgemeinerung der Delta-Regel, genannt *Backpropagation of Error*, zu:

$$\begin{aligned} \Delta w_{k,h} &= \eta o_k \delta_h \text{ mit} \\ \delta_h &= \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) \text{ } (h \text{ außen}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) \text{ } (h \text{ innen}) \end{cases} \end{aligned} \tag{5.37}$$

Im Unterschied zur Delta-Regel ist also die Behandlung des δ, je nachdem, ob es sich bei h um ein Ausgabe- oder aber inneres (also verdecktes) Neuron handelt, verschieden:

1. Ist h Ausgabeneuron, so gilt

$$\delta_{p,h} = f'_{\text{act}}(\text{net}_{p,h}) \cdot (t_{p,h} - y_{p,h}) \tag{5.38}$$

   Das Gewicht $w_{k,h}$ von k nach h wird also unter unserem Übungsmuster p proportional zu

   - ▷ Lernrate η,
   - ▷ Ausgabe $o_{p,k}$ des Vorgängerneurons k,
   - ▷ Gradient der Aktivierungsfunktion an der Stelle der Netzeingabe des Nachfolgerneurons $f'_{\text{act}}(\text{net}_{p,h})$ und
   - ▷ Differenz aus Teaching Input $t_{p,h}$ und Ausgabe $y_{p,h}$ des Nachfolgerneurons h

   geändert. *In diesem Fall arbeitet Backpropagation also auf zwei Neuronenschichten*, der Ausgabeschicht mit dem Nachfolgerneuron h und der Schicht davor mit dem Vorgängerneuron k.

2. Ist h inneres, verdecktes Neuron, so gilt

$$\delta_{p,h} = f'_{\text{act}}(\text{net}_{p,h}) \cdot \sum_{l \in L}(\delta_{p,l} \cdot w_{h,l}) \tag{5.39}$$

   An dieser Stelle sei ausdrücklich erwähnt, *dass Backpropagation nun auf drei Schichten arbeitet*. Hierbei ist das Neuron k der Vorgänger der zu ändernden Verbindung mit dem Gewicht $w_{k,h}$, das Neuron h der Nachfolger der zu ändernden Verbindung, und die Neurone l liegen in der Schicht *nach* dem Nachfolgerneuron. Das Gewicht $w_{k,h}$ von k nach h wird also unter unserem Übungsmuster p proportional zu

- ▷ Lernrate η,
- ▷ Ausgabe des Vorgängerneurons $o_{p,k}$,
- ▷ Gradient der Aktivierungsfunktion an der Stelle der Netzeingabe des Nachfolgerneurons $f'_{\text{act}}(\text{net}_{p,h})$,
- ▷ sowie, und hier liegt der Unterschied, aus der gewichteten Summe der Gewichtsveränderungen zu allen Neuronen, die h nachfolgen, $\sum_{l \in L}(\delta_{p,l} \cdot w_{h,l})$

geändert.

Definition 5.8 (Backpropagation)**.** Fassen wir die Formeln 5.38 auf der linken Seite und 5.39 auf der linken Seite zusammen, so erhalten wir folgende Gesamtformel für ***Backpropagation*** (die Bezeichner p werden der Übersichtlichkeit halber weggelassen):

$$\begin{aligned} \Delta w_{k,h} &= \eta o_k \delta_h \text{ mit} \\ \delta_h &= \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) \ (h \text{ außen}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) \ (h \text{ innen}) \end{cases} \end{aligned} \tag{5.40}$$

SNIPE: Eine online-Variante von Backpropagation ist in der Methode `trainBackpropagationOfError` der Klasse `NeuralNetwork` implementiert.

Offensichtlich ist also, dass Backpropagation zunächst die hinterste Gewichtsschicht direkt mit dem Teaching Input bearbeitet und sich anschließend ebenenweise unter Berücksichtigung der jeweils vorhergehenden Gewichtsänderungen weiter nach vorn arbeitet. *Der Teaching Input hinterlässt also Spuren in sämtlichen Gewichtsschichten.* Ich beschreibe hier gerade den ersten Teil (Delta-Regel) und zweiten Teil von Backpropagation (Verallgemeinerte Delta-Regel auf mehr Schichten) in einem Zug, was vielleicht der Sache, nicht jedoch der Forschung daran gerecht wird. Der erste Teil ist offensichtlich, das werden wir gleich im Rahmen einer mathematischen Spielerei sehen. *Zwischen dem ersten und zweiten, rekursiven Teil liegen jedoch Jahrzehnte an Entwicklungszeit und -arbeit*, denn wie bei vielen bahnbrechenden Erfindungen merkte man auch dieser erst *nach* der Entwicklung an, wie einleuchtend sie eigentlich ist.

5.4.2 Der mathematische Rückweg: Reduktion von Backpropagation auf Delta-Regel

Wie oben erläutert, ist die Delta-Regel ein Spezialfall von Backpropagation für einstufige Perceptrons und lineare Aktivierungsfunktionen – diesen Umstand möchte ich hier kurz näher erläutern und die Delta-Regel aus Backpropagation entwickeln, um

das Verständnis für beide Regeln noch etwas zu schärfen. Wir haben gesehen, dass Backpropagation durch

$$\Delta w_{k,h} = \eta o_k \delta_h \text{ mit}$$
$$\delta_h = \begin{cases} f'_{\text{act}}(\text{net}_h) \cdot (t_h - y_h) \ (h \text{ außen}) \\ f'_{\text{act}}(\text{net}_h) \cdot \sum_{l \in L}(\delta_l w_{h,l}) \ (h \text{ innen}) \end{cases} \tag{5.41}$$

definiert ist. Da wir sie nur für einstufige Perceptrons verwenden, fällt der zweite Teil von Backpropagation (heller dargestellt) ersatzlos weg, wir erhalten also:

$$\Delta w_{k,h} = \eta o_k \delta_h \text{ mit}$$
$$\delta_h = f'_{\text{act}}(\text{net}_h) \cdot (t_h - o_h) \tag{5.42}$$

Weiterhin wollen wir nur lineare Aktivierungsfunktionen verwenden, so dass f'_{act} (heller dargestellt) konstant ist. Konstanten lassen sich bekanntlich zusammenfassen, wir fassen also die konstante Ableitung f'_{act} und die (mindestens pro Lernzyklus konstante) Lernrate η (auch heller dargestellt) direkt in η zusammen. Es ergibt sich also:

$$\Delta w_{k,h} = \eta o_k \delta_h = \eta o_k \cdot (t_h - o_h) \tag{5.43}$$

Dies entspricht genau der Definition der Delta-Regel.

5.4.3 Die Wahl der Lernrate hat enormen Einfluss auf den Lernprozeß

Wie mittlerweile vielfach gesehen, ist die Gewichtsänderung in jedem Fall proportional zur Lernrate η. Die Wahl von η ist also sehr entscheidend für das Verhalten von Backpropagation und allgemein für Lernverfahren.

Definition 5.9 (Lernrate)**.** Die Geschwindigkeit und Genauigkeit eines Lernverfahrens ist immer steuerbar von und proportional zu einer ***Lernrate***, welche als η geschrieben wird.

Wird η zu groß gewählt, so sind die Sprünge auf der Fehlerfläche zu groß und es könnten z.B. enge Täler, also gute Werte, einfach übersprungen werden, zudem bewegt man sich sehr unkontrolliert über die Fehleroberfläche. Das Wunschmittel ist also ein kleines η, was aber einen riesigen, oft inakzeptablen Zeitaufwand mit sich bringen kann. Die Erfahrung zeigt, dass gute Werte für die Lernrate im Bereich

$$0.01 \leq \eta \leq 0.9$$

liegen. Die Wahl von η hängt maßgeblich von Problem, Netz und Trainingsdaten ab, so dass man kaum praktische Wahlhilfen geben kann. Beliebt ist jedoch, mit einem relativ großen η, z.B. 0.9, den Anfang des Lernens zu machen und η dann langsam bis auf z.B. 0.1 zu verringern, während für einfachere Probleme η oft einfach konstant gehalten werden kann.

5.4.3.1 Variation der Lernrate über die Zeit

Weiteres Stilmittel beim Training kann eine ***variable Lernrate*** sein: Eine große Lernrate lernt am Anfang gut, aber später nicht mehr genau, eine kleinere ist aufwändiger, lernt aber genauer. Also verringert man einmal oder mehrere Male die Lernrate um eine Größenordnung – während des Lernvorgangs.

Ein beliebter Fehler (der obendrein noch auf den ersten Blick sehr elegant wirkt) ist, die Lernrate kontinuierlich sinken zu lassen: Hier kommt es sehr leicht vor, dass der Abfall der Lernrate größer ist als die Steigung eines Hügels der Fehlerfunktion, die wir gerade erklimmen – was zur Folge hat, dass wir an dieser Steigung einfach hängen bleiben. Lösung: Lieber wie beschrieben die Lernrate stufenweise verringern.

5.4.3.2 Verschiedene Schichten – verschiedene Lernraten

Je weiter man sich während des Lernvorganges von der Ausgabeschicht wegbewegt, um so langsamer lernt Backpropagation – es ist also eine gute Idee, für die Gewichtsschichten nahe der Eingabeschicht eine größere Lernrate zu nehmen als für diejenigen nahe der Ausgabeschicht. Der Unterschied kann hier ruhig bis zu einer Größenordnung betragen.

5.5 Resilient Backpropagation ist eine Erweiterung von Backpropagation of Error

Gerade haben wir zwei Backpropagation-spezifische Eigenschaften angesprochen, die hin und wieder zum Problem werden können (zusätzlich zu denjenigen, die Gradientenabstiege ohnehin schon mit sich bringen): Zum einen kann der Benutzer von Backpropagation eine schlechte Lernrate wählen, und zum anderen lernt Backpropagation immer langsamer, je weiter die Gewichte von der Ausgabeschicht entfernt sind. Aus diesem Grund haben Martin Riedmiller et al. Backpropagation weiterentwickelt, und ihre Variante ***Resilient Backpropagation*** (kurz: ***Rprop***) getauft [RB93, Rie94]. Ich möchte Backpropagation und Rprop hier gegenüberstellen, ohne ausdrücklich eine Variante als „besser“ zu deklarieren. Bevor wir uns nun wirklich mit Formeln auseinandersetzen, wollen wir die zwei primären Ideen hinter Rprop (und ihre Folgen) erst einmal umgangssprachlich dem schon bekannten Backpropagation gegenüberstellen.

Lernrate: Backpropagation benutzt standardmäßig eine Lernrate η, die vom Benutzer gewählt wird, und für das ganze Netz gilt. Sie bleibt statisch, bis sie manuell geändert wird. Die Nachteile dieser Vorgehensweise haben wir schon erforscht. Rprop verfolgt hier einen komplett anderen Ansatz: Es gibt keine globale Lernrate. Erstens hat jedes einzelne Gewicht $w_{i,j}$ seine eigene Lernrate $\eta_{i,j}$, und zweitens werden diese Lernraten nicht vom Benutzer gewählt, sondern von Rprop selbst festgelegt. Drittens bleiben die Gewichtsänderungen nicht statisch, sondern werden von Rprop für jeden Zeitschritt angepasst. Um die zeitliche Änderung miteinzubeziehen, nennen wir sie korrekterweise $\eta_{i,j}(t)$. Dies ermöglich nicht nur gezielteres Lernen, auch das Problem des schichtweise verlangsamten Lernens wird auf elegante Weise gelöst.

Gewichtsänderung: In Backpropagation werden die Gewichte proportional zum Gradienten der Fehlerfunktion geändert. Das ist auf den ersten Blick wirklich intuitiv, allerdings übernehmen wir so jegliche Zerklüftung, die die Fehleroberfläche aufweist, mit in die Gewichtsänderung. Ob das immer sinnvoll ist, darf zumindest angezweifelt werden. Auch hier geht Rprop andere Wege: Der Betrag der Gewichtsänderung $\Delta w_{i,j}$ entspricht einfach direkt der zugehörigen, automatisch angepassten Lernrate $\eta_{i,j}$. So ist die Gewichtsänderung *nicht* proportional zum Gradienten, nur noch das Vorzeichen des Gradienten geht in die Gewichtsänderung mit ein. Bis jetzt wissen wir noch nicht, auf welche Weise die $\eta_{i,j}$ zur Laufzeit angepasst werden, aber es sei vorweggenommen, dass der Ergebnisprozess deutlich weniger zerklüftet aussieht als eine Fehlerfunktion.

Gegenüber Backprop wird also der Gewichts-Updateschritt ersetzt, und ein zusätzlicher Lernraten-Anpassungsschritt hinzugefügt. Wie werden diese Ideen nun genau umgesetzt?

5.5.1 Gewichtsänderungen sind nicht proportional zum Gradienten

Betrachten wir zunächst die Gewichtsänderung. Wir haben schon bemerkt, dass die gewichtsspezifischen Lernraten direkt als Beträge der Gewichtsänderungen für ihr jeweiliges Gewicht herhalten. Bleibt die Frage, woher das Vorzeichen kommt – und das ist eine Stelle, bei welcher der Gradient ins Spiel kommt. Wie schon bei der Herleitung von Backpropagation, leiten wir die Fehlerfunktion $\text{Err}(W)$ nach den einzelnen Gewichten $w_{i,j}$ ab und erhalten so Gradienten $\frac{\partial \text{Err}(W)}{\partial w_{i,j}}$. Jetzt kommt der große Unterschied: Anstatt den Betrag des Gradienten multiplikativ mit in die Gewichtsänderung einfließen zu lassen, betrachten wir nur das *Vorzeichen* des Gradienten. Der Gradient bestimmt also nicht mehr die Stärke, sondern nur noch die Richtung der Gewichtsänderung. Ist das Vorzeichen des Gradienten $\frac{\partial \text{Err}(W)}{\partial w_{i,j}}$ positiv, müssen wir das Gewicht $w_{i,j}$ verringern.

Vom Gewicht wird also $\eta_{i,j}$ abgezogen. Ist das Vorzeichen des Gradienten hingegen negativ, so bedarf das Gewicht einer Verstärkung, bekommt also $\eta_{i,j}$ addiert. Ist der Gradient genau 0, passiert einfach gar nichts. Wir gießen diese umgangssprachliche Beschreibung nun in eine Formel. Wir fügen jeweils ein (t) an, um darzustellen, dass alles im gleichen Zeitschritt passiert. Das ist unübersichtlicher, aber trotzdem wichtig, denn gleich werden wir noch eine weitere Formel betrachten, die über verschiedene Zeitschritte operiert. Dafür kürzen wir den Gradienten ab: $g = \frac{\partial \mathrm{Err}(W)}{\partial w_{i,j}}$.

Definition 5.10 (Gewichtsänderung in Rprop)**.**

$$\Delta w_{i,j}(t) = \begin{cases} -\eta_{i,j}(t), & \text{wenn } g(t) > 0 \\ +\eta_{i,j}(t), & \text{wenn } g(t) < 0 \\ 0 & \text{sonst.} \end{cases} \tag{5.44}$$

Wir wissen nun, wie die Gewichte an sich geändert werden - jetzt bleibt noch die Frage, auf welche Weise die Lernraten selbstständig angepasst werden. Zuletzt, wenn wir das Gesamtsystem verstanden haben, klären wir dann die verbleibenden Kleinigkeiten wie Initialisierungswerte und ein paar konkrete Konstanten.

5.5.2 Viele dynamisch angepasste Lernraten statt einer statischen

Um die Lernraten $\eta_{i,j}$ anzupassen, müssen wir wieder die zugehörigen Gradienten g betrachten, und zwar über zwei Zeitschritte hinweg: Den gerade vergangenen $(t-1)$ und den jetzigen (t). Wieder ist für uns nur das Vorzeichen des Gradienten wichtig, und wir müssen uns nun fragen: Was kann mit dem Vorzeichen über zwei Schritte hinweg passieren? Es kann gleich bleiben, und es kann wechseln.

Wechselt das Vorzeichen von $g(t-1)$ zu $g(t)$, so haben wir im Gradienten ein lokales Minimum übersprungen, das letzte Update ist also zu groß gewesen, folglich muss $\eta_{i,j}(t)$ im Vergleich zu dem vorherigen $\eta_{i,j}(t-1)$ verkleinert werden, die Suche muss genauer werden. Mathematisch ausgedrückt: Wir erhalten ein neues $\eta_{i,j}(t)$, in dem wir das alte $\eta_{i,j}(t-1)$ mit einer Konstante $\eta^{\downarrow}$ multiplizieren, die zwischen 1 und 0 liegt. In diesem Falle wissen wir ja, dass im letzten Zeitschritt $(t-1)$ etwas schiefgelaufen ist – also wird zusätzlich noch das Gewichtsupdate für das Gewicht $w_{i,j}$ im Zeitschritt (t) hart auf 0 gesetzt, also gar nicht erst durchgeführt (nicht in der folgenden Formel angegeben).

Bleibt das Vorzeichen aber gleich, kann eine (behutsame!) Vergrößerung von $\eta_{i,j}$ stattfinden, um über flache Bereiche der Fehlerfunktion hinwegzukommen. Hier erhalten wir

unser neues $\eta_{i,j}(t)$, in dem wir das alte $\eta_{i,j}(t-1)$ mit einer Konstante $\eta^{\uparrow}$ multiplizieren, die größer als 1 ist.

Definition 5.11 (Anpassung der Lernraten in Rprop)**.**

$$\eta_{i,j}(t) = \begin{cases} \eta^{\uparrow}\eta_{i,j}(t-1), & g(t-1)g(t) > 0 \\ \eta^{\downarrow}\eta_{i,j}(t-1), & g(t-1)g(t) < 0 \\ \eta_{i,j}(t-1) & \text{sonst.} \end{cases} \tag{5.45}$$

Achtung: Daraus folgt auch, dass Rprop ausschließlich für Offline-Lernen konzipiert ist, denn wenn die Gradienten nicht eine gewisse Kontinuität aufweisen, bremst das Lernverfahren auf niedrigstes Tempo ab (und verweilt dort). Wer online lernt, wechselt ja – salopp gesprochen – mit jeder neuen Epoche die Fehlerfunktion, da diese nur auf jeweils ein Trainingsmuster bezogen ist. Das geht zwar bei Backpropagation oft sehr gut und sogar sehr oft schneller als die Offline-Variante, weshalb es dort gerne eingesetzt wird. Es fehlt aber die saubere mathematische Motivation, und genau diese benötigen wir hier.

5.5.3 Uns fehlen noch ein paar Kleinigkeiten, um Rprop in der Praxis zu verwenden

Es bleiben noch ein paar kleinere Fragen offen, nämlich

1. Wie groß sind $\eta^{\uparrow}$ und $\eta^{\downarrow}$ (wie stark werden Lernraten verstärkt, bzw. abgeschwächt)?
2. Wie groß ist $\eta_{i,j}(0)$ (wie werden die gewichtsspezifischen Lernraten initialisiert)?[4]
3. Wie sind die oberen und unteren Grenzen $\eta_{\min}$ bzw. $\eta_{\max}$ für die $\eta_{i,j}$ gesetzt?

Die Antworten auf diese Fragen handeln wir nun mit kurzer Motivation ab. Der Initialisierungswert für die Lernraten sollte irgendwo in der Größenordnung der Gewichtsinitialisierung liegen, und so hat sich bewährt, $\eta_{i,j}(0) = 0.1$ zu setzen. Die Autoren der Rprop-Veröffentlichung beschreiben einleuchtenderweise, dass dieser Wert – solange er positiv gesetzt wird und keinen exorbitant hohen Betrag hat – eher unkritisch zu sehen ist, da er ohnehin schnell von der automatischen adaption überschrieben wird.

Ebenso unkritisch ist $\eta_{\max}$, für das ohne weitere mathematische Begründung ein Wert von 50 empfohlen und über die meiste weitere Literatur verwendet wird. Man kann

4 Protipp: Da die $\eta_{i,j}$ ausschließlich durch Multiplikation verändert werden, ist 0 als Initialisierungswert eher suboptimal :-)

diesen Parameter niedriger setzen, um ausschließlich sehr vorsichtige Updateschritte zu erlauben. Kleine Updateschritte sollten in jedem Fall erlaubt sein, also setzen wir $\eta_{\min} = 10^{-6}$.

Bleiben die Parameter $\eta^{\uparrow}$ und $\eta^{\downarrow}$. Fangen wir mit $\eta^{\downarrow}$ an: Wenn dieser Wert zum Einsatz kommt, haben wir ein Minimum übersprungen, von dem wir nicht genau wissen, wo auf der übersprungenen Strecke es liegt. Analog zur Vorgehensweise der binären Suche, wo das Zielobjekt ebenfalls oft übersprungen wird, gehen wir davon aus, es läge in der Mitte der übersprungenen Strecke. Also müssen wir die Lernrate halbieren, weswegen sich ein $\eta^{\downarrow} = 0.5$ kanonischerweise anbietet. Wenn der Wert $\eta^{\uparrow}$ zum Einsatz kommt, sollen Lernraten umsichtig vergrößert werden, hier können wir also nicht die binäre Suche generalisieren und einfach den Wert 2.0 verwenden, sonst besteht das Lernraten-Update nachher fast nur noch aus Richtungswechseln. Problemunabhängig hat sich ein Wert von $\eta^{\uparrow} = 1.2$ als erfolgsversprechend erwiesen, wobei leichte Änderungen die Konvergenzgeschwindigkeit nicht signifikant beeinflusst haben. So konnte auch dieser Wert einfach als Konstante gesetzt werden.

Mit fortschreitender Rechengeschwindigkeit der Computer ist eine immer größere Verbreitung von sehr tiefen Netzwerken (***Deep networks***) , also Netzwerken mit sehr vielen Schichten, zu beobachten. Für solche Netze ist Rprop dem originalen Backpropagation unbedingt vorzuziehen, weil Backprop, wie schon angedeutet, auf Gewichten fern der Ausgabeschicht sehr langsam lernt. Bei Problemen mit kleineren Schichtenzahlen würde ich empfehlen, das verbreitete Backpropagation (sowohl mit offline- als auch mit online-Lernen) und das weniger verbreitete Rprop zunächst gleichwertig zu testen.

SNIPE: Resilient Backpropagation wird in Snipe über die Methode `trainResilientBackpropagation` der Klasse `NeuralNetwork` unterstützt. Wahlweise kann man hier auch noch eine weitere Verbesserung zu Resilient Propagation zuschalten, die in dieser Arbeit jedoch nicht weiter behandelt wird. Für die verschiedenen Rprop-Parameter finden sich Getter und Setter.

5.6 Backpropagation wurde auch außerhalb von Rprop vielfach erweitert und variiert

Backpropagation ist vielfach erweitert worden – viele dieser Erweiterungen kann man einfach als optionale Features von Backpropagation implementieren, um größeren Testspielraum zu haben. Ich möchte im folgenden einige von ihnen kurz beschreiben.

5.6.1 Masseträgheit zum Lernprozeß hinzufügen

Angenommen, wir fahren auf Skiern einen steilen Hang hinab – was hindert uns, am Rande des Hangs zum Plateau sofort stehenzubleiben? Genau – der *Schwung*. Der *Momentum-Term* [RHW86b] sorgt bei Backpropagation dafür, dass der Schrittweite eine Art *Trägheitsmoment* (***Momentum***) hinzugefügt wird (Abb. 5.13 auf der rechten Seite), indem jeder neuen Gewichtsänderung immer ein Anteil der vorherigen Änderung hinzuaddiert wird:

$$(\Delta_p w_{i,j})_{\text{jetzt}} = \eta o_{p,i} \delta_{p,j} + \alpha \cdot (\Delta_p w_{i,j})_{\text{vorher}}$$

Diese Schreibweise dient natürlich nur dem besseren Verständnis; in der Regel wird, wie bereits durch den Zeitbegriff definiert, der Zeitpunkt des aktuellen Durchlaufs durch (t) bezeichnet, der vorherige Durchlauf wird dann durch $(t-1)$ gekennzeichnet, was man sukzessive fortführt. Wir kommen also zur formalen Definition des Momentum-Terms:

Definition 5.12 (Momentum-Term)**.** Die Variation von Backpropagation durch den ***Momentum-Term*** ist wie folgt definiert:

$$\Delta w_{i,j}(t) = \eta o_i \delta_j + \alpha \cdot \Delta w_{i,j}(t-1) \tag{5.46}$$

Wir beschleunigen also auf Plateaus (verhindert Quasi-Stillstand auf Plateaus) und bremsen auf zerklüfteten Flächen (gegen Oszillationen). Weiterhin kann man den Effekt der Trägheit über den Vorfaktor α variieren, übliche Werte befinden sich zwischen 0.6 und 0.9. Außerdem macht das Momentum den positiven Effekt möglich, dass unser Skifahrer in einem Minimum ein paar mal hin- und herpendelt, und schlussendlich in dem Minimum landet. Leider tritt trotz des schönen Aussehens im eindimensionalen der ansonsten seltene Fehler des Verlassens guter Minima durch den Momentum-Term häufiger auf – so dass auch hier wieder keine Patentlösung gegeben ist (wir gewöhnen uns ja langsam an diese Aussage).

5.6.2 Flat spot elimination verhindert, dass sich Neurone verfangen

Es ist zu beachten, dass sowohl beim *Tangens Hyperbolicus* sowie der *Fermifunktion* die Ableitung außerhalb unmittelbarer Nähe zu Θ fast 0 ist. Dieser Umstand führt dazu, dass sich Neurone nur schwer wieder aus den Grenzwerten der Aktivierung (*flat spots*) entfernen können , was die Lernzeit extrem verlängern kann. Diesem Problem kann durch Modifikation der Ableitung, z.B. Addition einer Konstanten (z.B. 0.1), begegnet werden, was als ***Flat spot elimination*** oder – umgangssprachlicher – ***Zuckern*** bezeichnet wird.

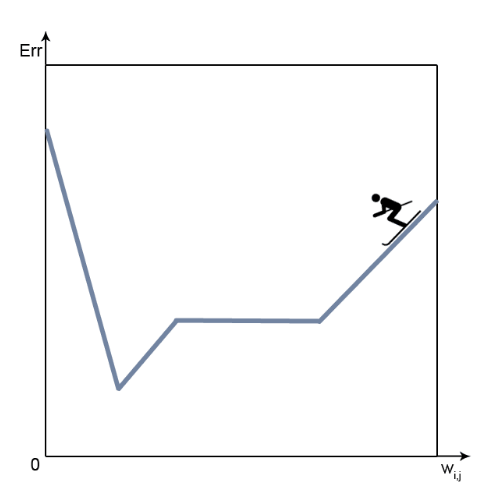


Abbildung 5.13: Wir möchten den Gradientenabstieg durchführen wie ein Skifahrer seine Abfahrt, der wohl kaum sofort an der Grenze zum Plateau anhalten wird.

Eine interessante Beobachtung ist, dass man auch schon reinen Konstanten als Ableitungen Erfolge erzielt hat [Fah88]. Auch die schon in Abschnitt 3.2.6 auf Seite 46 erwähnte schnelle Appoximation des Tangens Hyperbolicus von Anguita et al. benutzt in den äußeren Bereichen der ebenfalls approximierten Ableitung eine kleine Konstante.

5.6.3 Die zweite Ableitung kann mit einbezogen werden

Second Order Backpropagation nach David Parker [Par87] verwendet auch den zweiten Gradienten, also die zweite mehrdimensionale Ableitung der Fehlerfunktion, um genauere Schätzungen der korrekten $\Delta w_{i,j}$ zu erhalten. Höhere Ableitungen verbessern die Schätzungen nur noch selten. So braucht man weniger Trainingszyklen, diese sind aber weitaus rechenaufwändiger.

Bei Methoden höherer Ordnung im Allgemeinen werden weitere Ableitungen (also Hessesche Matrizen, da die Funktionen mehrdimensional sind) verwendet. Erwartungsgemäß reduzieren die Verfahren die Anzahl der Lernepochen, machen die einzelnen Epochen aber signifikant rechenaufwändiger – so dass am Ende die Lernzeit oft sogar länger ist als mit Backpropagation.

Das Lernverfahren ***Quickpropagation*** [Fah88] verwendet die zweite Ableitung der Fehlerfunktion und sieht die Fehlerfunktion lokal als eine Parabel an, deren Scheitelpunkt wir analytisch bestimmen und wohin wir direkt springen. Dieses Lernverfahren ist also ein Verfahren zweiter Ordnung. Es funktioniert natürlich nicht bei Fehleroberflächen, die nicht lokal durch eine Parabel approximierbar sind (ob das der Fall ist, kann man natürlich nicht immer direkt sagen).

5.6.4 Weight Decay: Große Gewichte können bestraft werden

Bei der Modifikation ***Weight Decay*** (zu Deutsch: Dämpfung der Gewichte) von Paul Werbos [Wer88] wird der Fehler um einen Term erweitert, der große Gewichte bestraft. Der Fehler unter Weight Decay

$$\mathrm{Err}_{\mathrm{WD}}$$

steigt also nicht nur mit dem eigentlichen Fehler, sondern auch mit dem Quadrat der Gewichte – was zur Folge hat, dass das Netz beim Lernen die Gewichte klein hält.

$$\mathrm{Err}_{\mathrm{WD}} = \mathrm{Err} + \underbrace{\beta \cdot \frac{1}{2} \sum_{w \in W} (w)^2}_{\text{Bestrafung}} \tag{5.47}$$

Dies ist von der Natur inspiriert, in der synaptische Gewichte ebenfalls nicht unendlich stark werden können. Klein gehaltene Gewichte sorgen außerdem häufig dafür, dass die Fehlerfunktion weniger starke Schwankungen beinhaltet, was das Lernen einfacher und kontrollierter macht.

Der Vorfaktor $\frac{1}{2}$ ist wieder aus einfacher Pragmatik heraus entstanden. Der Faktor β regelt die Stärke der Bestrafung: Werte von 0.001 bis 0.02 werden hier oft verwendet.

5.6.5 Das Netz zurechtstutzen: Pruning und Optimal Brain Damage

Wenn wir das Weight Decay lange genug durchgeführt haben und feststellen, dass bei einem Neuron im Eingabelayer alle Nachfolgegewichte Null oder fast Null sind, können wir das Neuron entfernen, haben ein Neuron und einige Gewichte verloren und reduzieren so die Chance, dass das Netz auswendig lernt. Dieser Vorgang wird als ***Pruning*** („Stutzen“) bezeichnet.

Solch ein Verfahren, unnötige Gewichte und Neurone zu detektieren und wegzustreichen, nennt sich ***Optimal Brain Damage*** [lCDS90]. Es sei hier nur kurz beschrieben:

Der Fehler pro Outputneuron setzt sich hierbei aus zwei konkurrierenden Termen zusammen. Während der eine wie gewohnt die Differenz zwischen Output und Teaching Input berücksichtigt, versucht der andere, ein Gewicht gegen 0 zu „pressen". Wird ein Gewicht nun stark benötigt, um den Fehler zu minimieren, gewinnt der erste Term – ist dies nicht der Fall, gewinnt der zweite. Neurone, die nur Nullgewichte besitzen, können zum Schluss wieder gestutzt werden.

Es gibt noch viele weitere Variationen von Backprop bzw. ganze Bücher eigens hierüber – da mein Ziel aber ist, einen Überblick über Neuronale Netze zu bieten, möchte ich hier nur die obigen als Anstoß zum Weiterlesen nennen.

Es ist bei manchen dieser Erweiterungen offensichtlich, dass sie nicht nur bei FeedForward-Netzen mit Backpropagation-Lernverfahren angewendet werden können.

Wir haben nun Backpropagation und die FeedForward-Topologie kennen gelernt – nun fehlt uns noch Erfahrung, wie wir ein Netz aufbauen. Diese Erfahrung ist im Rahmen dieser Arbeit natürlich nicht vermittelbar, und um ein wenig davon zu erwerben, empfehle ich nun, ein paar der Beispielproblemstellungen aus Abschnitt 4.6 anzugehen.

5.7 Wie fängt man an? Initialkonfiguration eines Multilayerperzeptrons

Nachdem wir jetzt das Lernverfahren Backpropagation of Error behandelt haben und wissen, wie wir ein einmal vorhandenes Netz trainieren, ist es noch sinnvoll zu betrachten, wie wir denn überhaupt an so ein Netz gelangen können.

5.7.1 Anzahl der Schichten: Zwei oder drei sind oft genug, mehr werden aber auch benutzt

Fangen wir mit dem trivialen Umstand an, dass ein Netz eine Schicht Inputneurone und eine Schicht Outputneurone haben sollte, was uns zu mindestens zwei Schichten führt.

Weiterhin benötigen wir, wie wir bereits während unserer Untersuchung der linearen Separierbarkeit erfahren haben, mindestens eine versteckte Schicht Neurone, falls unser Problem nicht linear separierbar ist (wie wir gesehen haben, ist das wahrscheinlich).

Es kann, wie schon gesagt, mathematisch bewiesen werden, dass dieses MLP mit einer versteckten Neuronenschicht bereits beliebige Funktionen beliebig genau approximieren kann[5] – doch müssen wir nicht nur die ***Repräsentierbarkeit*** eines Problems durch ein Perceptron betrachten, sondern auch die ***Lernbarkeit***. Repräsentierbarkeit bedeutet, dass ein Perceptron eine Abbildung grundsätzlich realisieren kann – Lernbarkeit bezeichnet aber, dass wir sie ihm auch beibringen können.

Insofern zeigt uns die Erfahrung, dass zwei versteckte Neuronenschichten bzw. drei trainierbare Gewichtsschichten für die Realisierung eines Problems sehr nützlich sein können, da viele Probleme zwar durchaus von einer versteckten Schicht repräsentiert werden können, jedoch leider nur schwer lernbar sind.

Jede weitere Schicht erzeugt auch weitere Nebenminima der Fehlerfunktion, was bei der Wahl der Schichtenzahl beachtet werden sollte. Ein erfolgversprechender Weg ist also zusammenfassend, es erst mit einer versteckten Schicht zu probieren. Wenn das nicht klappt, versucht man es mit zweien, und erst wenn das nicht funktioniert auf mehr Schichten auszuweichen. Dennoch werden mit zunehmender Rechenkraft der Computer für manche Probleme bereits ***Deep networks*** mit sehr vielen Schichten erfolgreich angewandt.

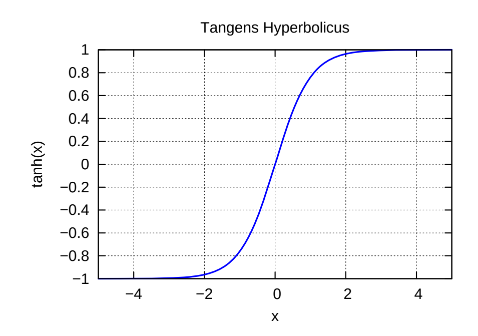
5.7.2 Anzahl der Neurone sollte getestet werden

Die Zahl der Neurone (abseits von Eingabe- und Ausgabeschicht, die Anzahl der Eingabe- und Ausgabeneurone ist ja durch die Problemstellung bereits fest definiert) entspricht grundsätzlich der Zahl der freien Parameter des zu repräsentierenden Problems.

Da wir ja schon die Netzkapazität in Bezug auf Auswendiglernen oder eine zu ungenaue Problemrepräsentation erforscht haben, ist klar, dass unser Ziel *so wenig wie möglich*, aber *so viel wie nötig* freie Parameter sind.

Wie wir aber auch wissen, gibt es keine Patentformel, wie viele Neurone zu verwenden sind – die sinnvollste Herangehensweise besteht also darin, zunächst mit wenigen Neuronen zu trainieren und so lange neue Netze mit mehr Neuronen zu trainieren, wie das Ergebnis noch signifikant verbessert und vor allem die Generalisierungsleistung nicht beeinträchtigt wird (*Bottom-Up-Ansatz*).

5 Achtung: Wir haben keine Aussage über die Neuronenanzahl in der versteckten Schicht gemacht, nur über die theoretische Möglichkeit.



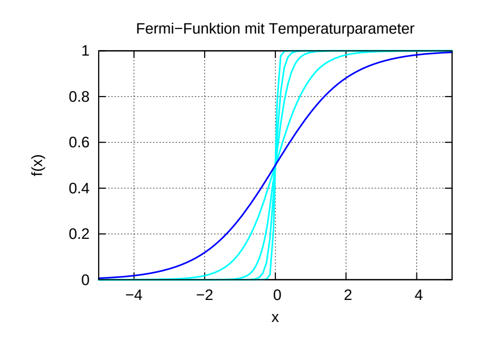


Abbildung 5.14: Zur Erinnerung noch einmal die Darstellung des Tangens Hyperbolicus (links) und der Fermifunktion (rechts). Die Fermifunktion wurde um einen Temperaturparameter erweitert. Die ursprüngliche Fermifunktion ist hierbei dunkel herausgestellt, die Temperaturparameter bei den modifizierten Fermifunktionen betragen von außen nach innen (aufsteigend geordnet nach Anstieg) $\frac{1}{2}$, $\frac{1}{5}$, $\frac{1}{10}$ und $\frac{1}{25}$.

5.7.3 Wahl der Aktivierungsfunktion

Ein weiterer sehr wichtiger Parameter für die Informationsverarbeitungsweise eines Neuronalen Netzes ist die ***Wahl der Aktivierungsfunktion***. Für die Inputneuronen steht die Aktivierungsfunktion ja fest, da sie nicht informationsverarbeitend sind.

Eine Frage, die man sich zunächst stellen kann, ist, ob man überhaupt in der versteckten und der Ausgabeschicht die gleiche Aktivierungsfunktion verwenden möchte – niemand hindert uns daran, hier zu variieren. In aller Regel ist die Aktivierungsfunktion aber für alle versteckten Neurone untereinander gleich, ebenso für die Outputneurone.

Für Aufgaben der ***Funktionsapproximation*** hat es sich als sinnvoll erwiesen, als Aktivierungsfunktion der versteckten Neurone den Tangens Hyperbolicus (linker Teil der Abb. 5.14) zu verwenden, während eine lineare Aktivierungsfunktion in der Ausgabe verwendet wird – letzteres ist unbedingt erforderlich, damit wir kein begrenztes Ausgabeintervall erzeugen. Da die Outputschicht im Gegensatz zur ebenfalls linearen Inputschicht Schwellenwerte besitzt, ist sie trotzdem informationsverarbeitend. Lineare Aktivierungsfunktionen in der Ausgabe können aber auch für riesige Lernschritte sorgen, und dafür, dass man gute Minima in der Fehleroberfläche überspringt. Dies kann verhindert werden, indem man die Lernrate an der Ausgabeschicht auf sehr kleine Werte setzt.

Für Aufgaben der ***Mustererkennung***[6] ist ein unbegrenztes Ausgabeintervall nicht unbedingt erforderlich. Verwendet man überall den Tangens Hyperbolicus, so ist das Ausgabeintervall etwas größer. Die Fermifunktion (rechter Teil der Abb. 5.14 auf der vorangehenden Seite) hat im Gegensatz zum Tangens Hyperbolicus weit vor dem Schwellenwert (wo ihr Ergebnis nahe 0 ist) kaum Möglichkeiten etwas zu lernen. Hier ist allerdings wieder viel Ermessensspielraum bei der Wahl der Aktivierungsfunktion gegeben. Sigmoide Funktionen haben allgemein aber den Nachteil, dass sie weit weg von ihrem Schwellenwert kaum noch etwas lernen, wenn man das Netz nicht etwas modifiziert.

5.7.4 Gewichte sollten klein und zufällig initialisiert werden

Die Initialisierung der Gewichte ist nicht so trivial wie man vielleicht denken mag: Initialisiert man sie einfach mit 0, wird gar keine Gewichtsänderung stattfinden. Initialisiert man sie alle mit demselben Wert, werden sie im Training immer gleichermaßen geändert. Die einfache Lösung dieses Problems nennt man ***Symmetry Breaking***. So wird die Initialisierung der Gewichte mit kleinen, zufälligen Werten bezeichnet. Als Bereich für die Zufallswerte könnte man das Intervall $[-0.5; 0.5]$ verwenden, jedoch ohne die Null oder Werte, die sehr nah bei Null liegen. Diese Zufallsinitialisierung hat einen schönen Nebeneffekt, nämlich dass der Durchschnitt der Netzeingaben wahrscheinlich nahe 0 ist. Dies ist nützlich, denn bei den gängigen Aktivierungsfunktionen liegt die 0 im Bereich der stärksten Ableitung der Aktivierungsfunktion, was kräftige Lernimpulse direkt zu Beginn des Lernens ermöglicht.

SNIPE: In Snipe werden die Gewichte zufällig initialisiert, falls eine Synapseninitialisierung gewünscht ist. Den maximalen Absolutbetrag eines Gewichts kann man in einem `NeuralNetworkDescriptor` mit der Methode `setSynapseInitialRange` festlegen.

5.8 Das 8-3-8-Kodierungsproblem und verwandte Probleme

Das 8-3-8-Kodierungsproblem ist ein Klassiker unter den Testtrainingsproblemen für Multilayerperceptrons. Wir besitzen in unserem MLP eine Eingabeschicht von acht Neuronen $i_1, i_2, \ldots, i_8$, eine Ausgabeschicht von acht Neuronen $\Omega_1, \Omega_2, \ldots, \Omega_8$, und eine versteckte Schicht von drei Neuronen. Dieses Netz repräsentiert also eine Funktion $\mathbb{B}^8 \to \mathbb{B}^8$. Die Trainingsaufgabe ist nun, dass, wenn in ein Neuron i_j der Wert 1

6 Mustererkennung wird in der Regel als Spezialfall der Funktionsapproximation mit wenigen diskreten Ausgabemöglichkeiten gesehen.

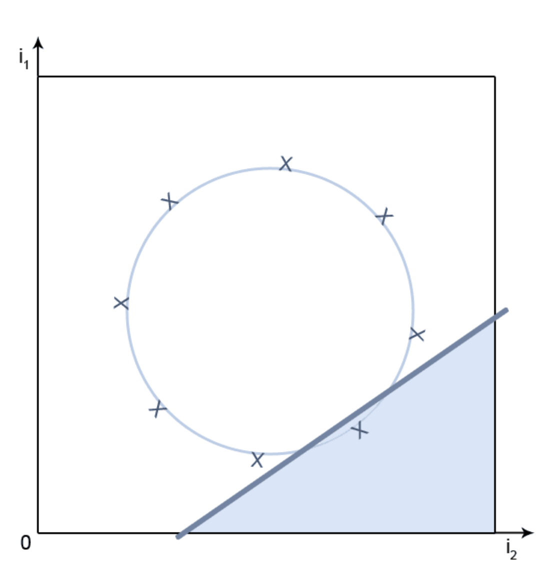


Abbildung 5.15: Skizze der Funktionsweise der Kodierung eines 8-2-8 Netzes. Die Punkte repräsentieren die Vektoren der Aktivierung der beiden inneren Neuronen. Wie Sie sehen, lassen sich durchaus Aktivierungsformationen finden, so dass jeder der Punkte durch eine Gerade vom Rest der Punkte separierbar ist. Diese Separierung ist im Bild für einen der Punkte exemplarisch durchgeführt.

eingegeben wird, genau im Neuron Ω_j der Wert 1 ausgegeben wird (es soll jeweils nur ein Neuron aktiviert werden, dies führt uns zu 8 Trainingsbeispielen).

Sie werden bei Analyse des trainierten Netzes sehen, dass das Netz mit den 3 versteckten Neuronen eine Art Binärkodierung repräsentiert und die obige Abbildung so möglich ist (mutmaßliche Trainingszeit hierfür sind $\approx 10^4$ Epochen). Wir haben mit unserem Netz also eine Maschine gebaut, die Daten kodiert und diese anschließend wieder dekodiert.

Analog hierzu kann man ein 1024-10-1024 Kodierungsproblem trainieren – doch geht das auch effizienter? Kann es beispielsweise ein 1024-9-1024 oder ein 8-2-8-Kodierungsnetz geben?

Ja, selbst das geht, da das Netz nicht auf binäre Kodierungen angewiesen ist: Ein 8-2-8-Netz funktioniert also für unsere Problemstellung. Die Kodierung, die dieses Netz realisiert, ist allerdings schwieriger zu durchschauen (Abb. 5.15) und es muss auch wesentlich länger trainiert werden.

Ein 8-1-8-Netz funktioniert nicht mehr, da die Möglichkeit vorhanden sein muss, dass die Ausgabe eines Neurons von einem anderen ausgeglichen wird, und bei nur einem versteckten Neuron natürlich kein Ausgleichsneuron vorhanden ist.

SNIPE: Die statische Methode `getEncoderSampleLesson` der Klasse `TrainingSampleLesson` erlaubt es, einfache Trainingsamples für derartige Encoderprobleme beliebiger Dimensionalität zu generieren.

Übungsaufgaben

Aufgabe 8. Ein 2-15-15-2-MLP soll durch ein MLP mit nur einer einzigen verdeckten Schicht, aber gleich vielen Gewichten ersetzt werden. Berechnen Sie, wieviele Neurone dieses Netz in seiner verdeckten Schicht hat. Hinweis: Vergessen Sie das BIAS-Neuron nicht.

Aufgabe 9. In Abb. 5.4 auf Seite 94 sehen Sie jeweils ein kleines Netz für die Booleschen Funktionen `AND` und `OR`. Schreiben Sie Tabellen, die sämtliche Berechnungsgrößen in den Neuronalen Netzen beinhalten (z.B. Netzeingabe, Aktivierungen, etc). Exerzieren Sie die vier möglichen Eingaben der Netze durch und notieren Sie die Werte dieser Größen für die jeweiligen Eingaben. Verfahren Sie in gleicher Weise für `XOR`-Netz (Abb. 5.9 auf Seite 106).

Aufgabe 10.

1. Nennen Sie alle Booleschen Funktionen $\mathbb{B}^3 \to \mathbb{B}^1$, welche linear separierbar sind, bzw. charakterisieren Sie sie genau.
2. Nennen Sie diejenigen, die es nicht sind, bzw. charakterisieren Sie sie genau.

Aufgabe 11. Ein einfaches 2-1-Netz soll mittels *Backpropagation of Error* und $\eta = 0.1$ mit einem einzigen Muster trainiert werden. Prüfen Sie, ob der Fehler

$$\text{Err} = \text{Err}_p = \frac{1}{2}(t - y)^2$$

konvergiert und wenn ja, zu welchem Wert. Wie sieht die Fehlerkurve aus? Das Muster (p, t) sei definiert zu $p = (p_1, p_2) = (0.3, 0.7)$ und $t_\Omega = 0.4$. Initialisieren Sie die Gewichte zufällig im Intervall $[1; -1]$.

Aufgabe 12. Ein einstufiges Perceptron mit zwei Inputneuronen, Biasneuron und binärer Schwellenwertfunktion als Aktivierungsfunktion trennt den zweidimensionalen Raum durch eine Gerade g in zwei Teile. Berechnen Sie für ein solches Perceptron

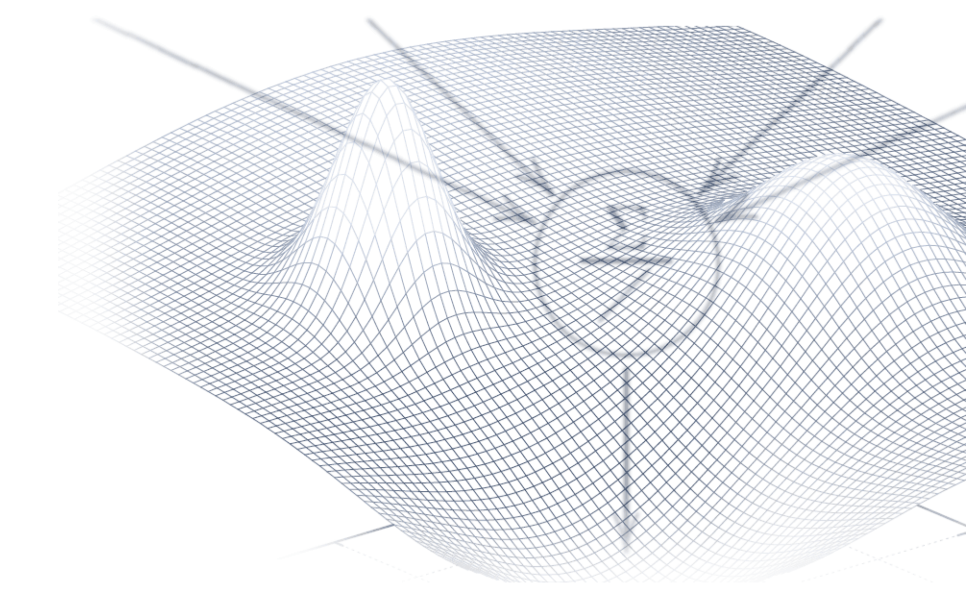
analytisch einen Satz Gewichtswerte, so dass die folgende Menge P der 6 Muster der Form (p_1, p_2, t_Ω) mit $\varepsilon \ll 1$ richtig klassifiziert wird.

$$\begin{aligned} P = \{ & (0, 0, -1); \\ & (2, -1, 1); \\ & (7 + \varepsilon, 3 - \varepsilon, 1); \\ & (7 - \varepsilon, 3 + \varepsilon, -1); \\ & (0, -2 - \varepsilon, 1); \\ & (0 - \varepsilon, -2, -1)\} \end{aligned}$$

Aufgabe 13. Berechnen Sie einmal und nachvollziehbar den Vektor ΔW sämtlicher Gewichtsänderungen mit dem Verfahren *Backpropagation of Error* mit $\eta = 1$. Gegeben Sei ein 2-2-1-MLP mit Biasneuron, das Muster sei definiert zu

$$p = (p_1, p_2, t_\Omega) = (2, 0, 0.1).$$

Die Initialwerte der Gewichte sollen bei allen Gewichten, welche Ω als Ziel haben, 1 betragen – sämtliche anderen Gewichte sollten den Initialwert 0.5 besitzen. Was fällt an den Änderungen auf?



Kapitel 6

Radiale Basisfunktionen

RBF-Netze nähern Funktionen an, indem sie Gaußglocken strecken, stauchen und anschließend räumlich versetzt aufsummieren. Beschreibung ihrer Funktion und ihres Lernvorganges, Gegenüberstellung mit Multilayerperceptrons.

Radiale Basisfunktionennetze (RBF-Netze) nach Poggio und Girosi [PG89] sind ein Paradigma Neuronaler Netze, welches deutlich später entstand als das der Perceptrons. Sie sind wie Perceptrons schichtartig aufgebaute Netze, allerdings in diesem Fall mit exakt drei Schichten, also nur einer einzigen Schicht versteckter Neurone.

Wie Perceptrons besitzen die Netze eine reine FeedForward-Struktur und Vollverknüpfung zwischen den Schichten, und auch hier trägt die Inputschicht nicht zur Informationsverarbeitung bei. Auch sind die RBF-Netze wie MLPs universelle Funktionsapproximatoren.

Bei allen Gemeinsamkeiten: Was unterscheidet die RBF-Netze nun von den Perceptrons? Es ist die Informationsverarbeitung selbst bzw. die Berechnungsvorschriften innerhalb der Neurone, welche nicht in der Inputschicht liegen. Wir werden also gleich eine bis jetzt völlig unbekannte Art Neurone neu definieren.

6.1 Bestandteile und Aufbau eines RBF-Netzes

Wir wollen nun zunächst einige Begriffe rund um die RBF-Netze erst umgangssprachlich betrachten und danach definieren.

Ausgabeneurone in einem RBF-Netz enthalten nur die Identität als Aktivierungsfunktion und eine gewichtete Summe als Propagierungsfunktion. Sie machen also nichts weiter, als alles, was in sie eingegeben wird, aufzusummieren und die Summe auszugeben.

Versteckte Neurone heißen auch RBF-Neurone (so wie die Schicht, in der sie sich befinden, auch RBF-Schicht genannt wird). Jedes versteckte Neuron erhält als Propagierungsfunktion eine Norm, welche den Abstand zwischen der Eingabe in das Netz und dem sogenannten Ort des Neurons (Zentrum) errechnet. Diese wird in eine radiale Aktivierungsfunktion eingegeben, die die Aktivierung des Neurons berechnet und ausgibt.

Definition 6.1 (RBF-Eingabeneuron)**.** Die Definition und Darstellung ist identisch mit der Eingabeneuron-Definition 5.1 auf Seite 90.

Definition 6.2 (Zentrum eines RBF-Neurons)**.** Das ***Zentrum*** c_h eines RBF-Neurons h ist der Punkt im Eingaberaum, in dem das RBF-Neuron angesiedelt ist. Je näher der Eingabevektor am Zentrumsvektor eines RBF-Neurons liegt, umso höher ist in der Regel seine Aktivierung.

Definition 6.3 (RBF-Neuron)**.** Die sogenannten ***RBF-Neurone*** h besitzen eine Propagierungsfunktion f_{prop}, welche den *Abstand* zwischen dem *Zentrum* c_h eines Neurons und dem Eingabevektor y feststellt. Dieser Abstand repräsentiert dann die Netzeingabe. Die Netzeingabe wird dann durch eine Radialbasisfunktion f_{act} geschickt, welche die Aktivierung bzw. Ausgabe des Neurons ausgibt. RBF-Neurone werden durch das Symbol $\frac{||c,x||}{\text{Gauß}}$ dargestellt.

Definition 6.4 (RBF-Ausgabeneuron)**.** ***RBF-Ausgabeneurone*** Ω besitzen die gewichtete Summe als Propagierungsfunktion f_{prop}, und die Identität als Aktivierungsfunktion f_{act}. Wir stellen sie durch das Symbol $\frac{\Sigma}{/}$ dar.

Definition 6.5 (RBF-Netz)**.** Ein ***RBF-Netz*** besitzt exakt drei Schichten in der folgenden Reihenfolge: Die Eingabeschicht aus Eingabeneuronen, die versteckte Schicht (auch RBF-Schicht genannt) aus RBF-Neuronen und die Ausgabeschicht aus RBF-Ausgabeneuronen. Jede Schicht ist mit der nächsten vollverknüpft, ShortCuts existieren nicht (Abb. 6.1 auf der rechten Seite) – es handelt sich also um eine reine FeedForward-Topologie. Die Verbindungen zwischen Eingabeschicht und RBF-Schicht sind ungewichtet, leiten die Eingabe also nur weiter. Die Verbindungen zwischen RBF- und Ausgabeschicht sind gewichtet. Die ursprüngliche Definition eines RBF-Netzes bezog sich auf nur ein Ausgabeneuron, analog zu den Perceptrons ist aber klar, dass sich

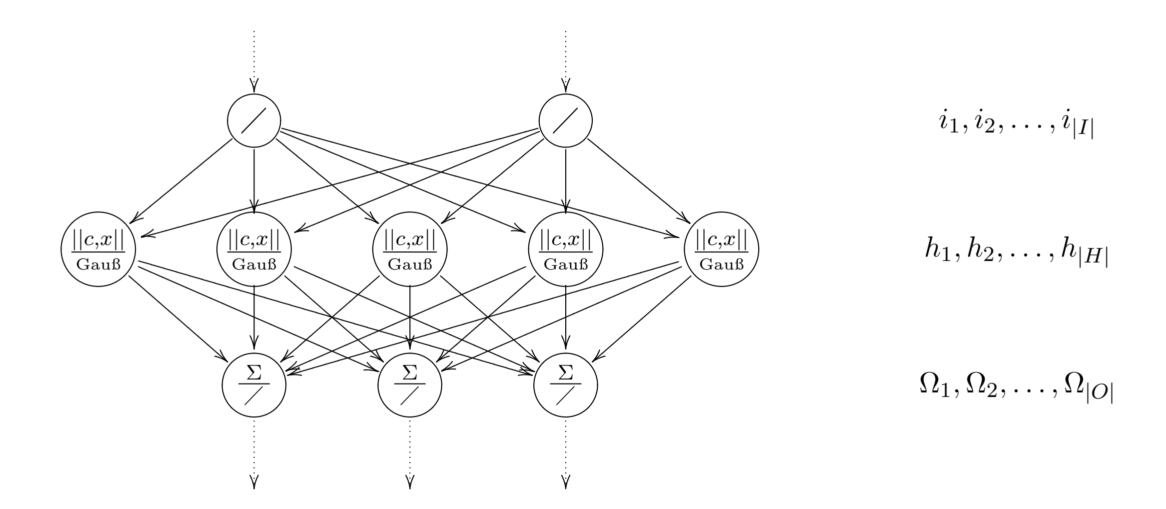


Abbildung 6.1: Ein beispielhaftes RBF-Netz mit zwei Eingabeneuronen, fünf versteckten Neuronen und drei Ausgabeneuronen. Die Verbindungen zu den versteckten Neuronen besitzen keine Gewichte, sie leiten die Eingabe nur weiter. Rechts der Skizze ist die Benennung der Neurone zu finden, welche sich analog zu der bekannten Benennung der Neurone im MLP verhält: Inputneurone heißen i, versteckte Neurone h, Ausgabeneurone Ω. Die zugehörigen Mengen bezeichnen wir mit I, H und O.

dies verallgemeinern lässt. Ein Biasneuron kennt das RBF-Netz nicht. Wir wollen die Menge der Eingabeneurone mit I, die Menge der versteckten Neurone mit H und die Menge der Ausgabeneurone mit O bezeichnen.

Die inneren Neurone heißen daher Radiale Basisneurone, weil aus deren Definition direkt folgt, dass alle Eingabevektoren, welche den gleichen Abstand vom Zentrum eines Neurons haben, auch den gleichen Ausgabewert produzieren (Abb. 6.2 auf der folgenden Seite).

6.2 Informationsverarbeitung eines RBF-Netzes

Die Frage ist nun, was durch dieses Netz realisiert wird und wo der Sinn liegt. Gehen wir doch einmal das RBF-Netz von oben nach unten durch: Der Input wird durch die

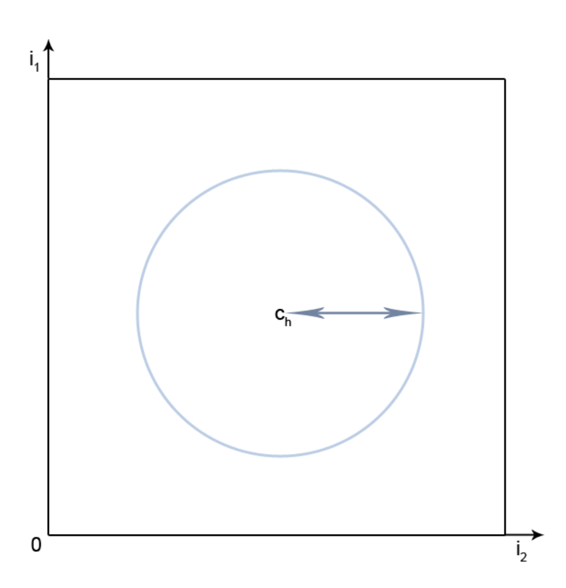


Abbildung 6.2: Sei c_h das Zentrum eines RBF-Neurons h. Dann liefert die Aktivierungsfunktion $f_{\text{act}h}$ für alle Eingaben, welche auf dem Kreis liegen, den gleichen Wert.

ungewichteten Verbindungen in ein RBF-Neuron eingebracht. Dieses schickt den Inputvektor durch eine Norm, so dass ein Skalar herauskommt. Dieser Skalar (der aufgrund der Norm übrigens nur positiv sein kann) wird durch eine Radiale Basisfunktion, also zum Beispiel eine Gaußglocke (Abb. 6.3 auf der rechten Seite) verarbeitet.

Die Ausgabewerte der verschiedenen Neurone der RBF-Schicht bzw- der verschiedenen Gaußglocken werden nun in der dritten Schicht aufsummiert: Faktisch werden, auf den ganzen Eingaberaum bezogen, also Gaußglocken aufsummiert.

Stellen wir uns vor, wir haben ein zweites, drittes und viertes RBF-Neuron und daher insgesamt vier unterschiedlich lokalisierte Zentren. Jedes dieser Neurone misst nun einen anderen Abstand von der Eingabe zum eigenen Zentrum und liefert de facto selbst bei gleicher Gaußglocke andere Werte. Da diese zum Schluß in der Ausgabeschicht nur aufkumuliert werden, ist leicht einsehbar, dass man durch Zerren, Stauchen und Verschieben von Gaußglocken und durch das anschließende Aufkumulieren jede beliebige Oberfläche modellieren kann. Die Entwicklungsterme für die Superposition der Gaußglocken liegen hierbei in den Gewichten der Verbindungen von RBF-Schicht zu Outputschicht.

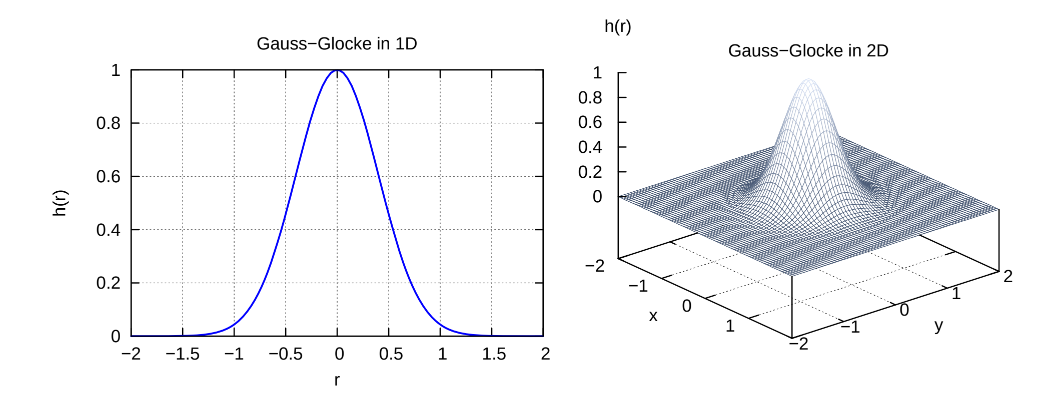


Abbildung 6.3: Zwei einzelne Gaußglocken im Ein- und Zweidimensionalen. In beiden Fällen gilt $\sigma = 0.4$ und in beiden Fällen ist das Zentrum der Gaußglocke im Nullpunkt. Der Abstand r zum Zentrum $(0, 0)$ berechnet sich schlicht aus dem Satz des Pythagoras: $r = \sqrt{x^2 + y^2}$.

Die Netzarchitektur bietet weiterhin die Möglichkeit, Höhe und Breite der Gaußglocken frei zu bestimmen oder zu trainieren – was dieses Netzparadigma noch vielfältiger macht. Methoden und Vorgehensweisen hierzu werden wir noch kennenlernen.

6.2.1 RBF-Neurone verarbeiten Information durch Normen und Radialbasisfunktionen

Nehmen wir zunächst ein einfaches 1-4-1-RBF-Netz als Beispiel. Es ist hier klar, dass wir eine eindimensionale Ausgabe erhalten werden, die wir als Funktion darstellen können (Abb. 6.4 auf der folgenden Seite). Das Netz besitzt weiterhin Zentren $c_1, c_2, \ldots, c_4$ der vier inneren Neurone $h_1, h_2, \ldots, h_4$, und somit Gaußglocken, die zum Schluss im Ausgabeneuron Ω aufsummiert werden. Das Netz besitzt auch vier Werte $\sigma_1, \sigma_2, \ldots, \sigma_4$, welche die Breite der Gaußglocken beeinflussen. Die Höhe der Gaußglocke wird hingegen von den nachfolgenden Gewichten beeinflusst, da damit die einzelnen Ausgabewerte der Glocken multipliziert werden.

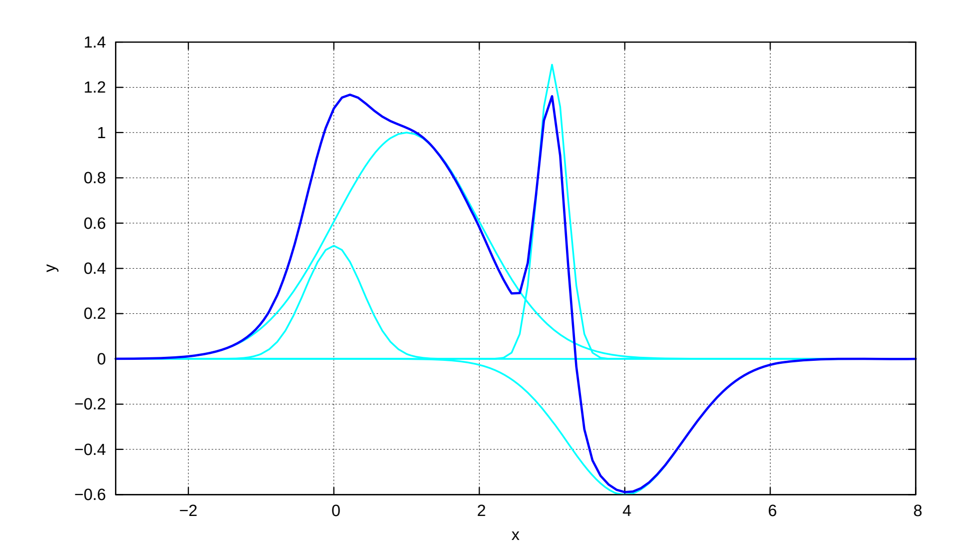


Abbildung 6.4: Vier verschiedene durch RBF-Neurone gebildete Gaußglocken im Eindimensionalen werden von einem Outputneuron des RBF-Netzes aufsummiert. Die Gaußglocken haben verschiedene Höhen, Breiten und Orte. Die Zentren $c_1, c_2, \ldots, c_4$ lagen bei $0, 1, 3, 4$, die Breiten $\sigma_1, \sigma_2, \ldots, \sigma_4$ bei $0.4, 1, 0.2, 0.8$. Ein Beispiel für den zweidimensionalen Fall findet sich in Abb. 6.5 auf der rechten Seite.

Da wir eine *Norm* zum Berechnen des Abstands des Inputvektors zum Neuronenzentrum eines Neurons h verwenden, haben wir verschiedene Wahlmöglichkeiten: Oft wird der *Euklidische Abstand* zur Abstandsberechnung gewählt:

$$r_h = ||x - c_h|| \tag{6.1}$$

$$= \sqrt{\sum_{i \in I} (x_i - c_{h,i})^2} \tag{6.2}$$

Wir erinnern uns: Mit x haben wir den Eingabevektor benannt. Hierbei durchläuft der Index i die Eingabeneurone und damit die Komponenten des Eingabevektors und des Neuronenzentrums. Wie wir sehen, bildet der Euklidische Abstand die Quadrate der Differenzen aller Vektorkomponenten, summiert sie auf und zieht aus der Summe die Wurzel, was im zweidimensionalen dem Satz des Pythagoras gleich kommt. Aus der Definition einer Norm folgt direkt, dass der Abstand nur positiv sein kann, weswegen wir genaugenommen den positiven Teil der Aktivierungsfunktion verwenden. Übrigens sind auch andere Aktivierungsfunktionen als die Gaußglocke möglich, in aller Regel werden Funktionen gewählt, die im Intervall $[0; \infty]$ monoton abfallen.

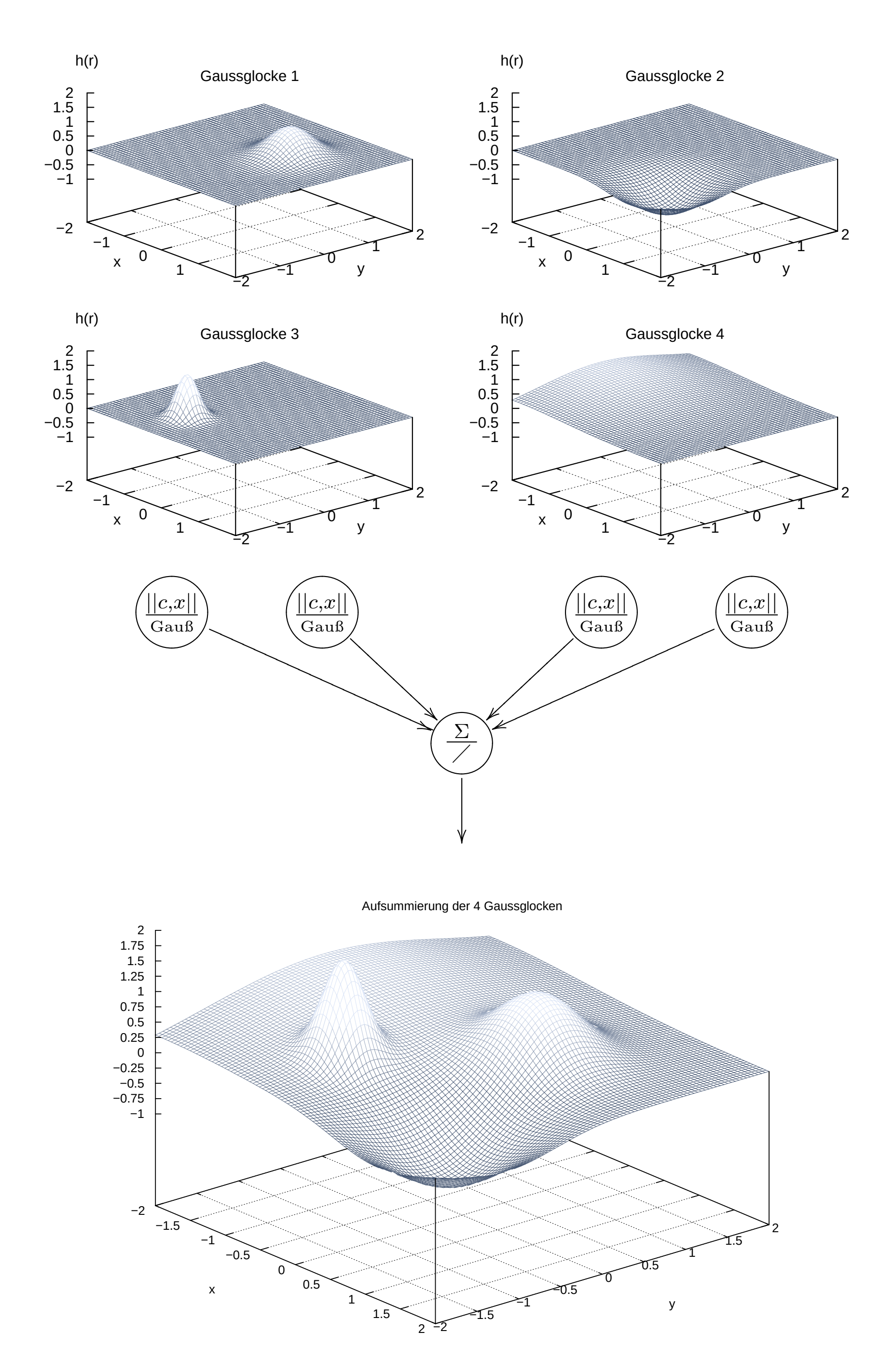


Abbildung 6.5: Vier verschiedene durch RBF-Neurone gebildete Gaußglocken im Zweidimensionalen werden von einem Outputneuron des RBF-Netzes aufsummiert. Für den Abstand gilt wieder $r = \sqrt{x^2 + y^2}$. Die Höhen w, Breiten σ und Zentren $c = (x, y)$ sind: $w_1 = 1, \sigma_1 = 0.4, c_1 = (0.5, 0.5)$, $w_2 = -1, \sigma_2 = 0.6, c_2 = (1.15, -1.15)$, $w_3 = 1.5, \sigma_3 = 0.2, c_3 = (-0.5, -1)$, $w_4 = 0.8, \sigma_4 = 1.4, c_4 = (-2, 0)$.

Nachdem wir nun den ***Abstand*** r_h des Inputvektors x zum Zentrum c_h des RBF-Neurons h kennen, muss dieser Abstand durch die Aktivierungsfunktion f_{act} geschickt werden – wir verwenden hier, wie schon gesagt, eine Gaußglocke:

$$f_{\text{act}}(r_h) = \mathrm{e}^{\left(\frac{-r_h^2}{2\sigma_h^2}\right)} \tag{6.3}$$

Es ist klar, dass sowohl Zentrum c_h als auch die Breite σ_h als Bestandteil der Aktivierungsfunktion f_{act} gesehen werden können und nach dieser Ansicht die Aktivierungsfunktionen nicht alle mit f_{act} bezeichnet werden dürften. Eine Lösung wäre, die Aktivierungsfunktionen nach dem Muster $f_{\text{act}1}, f_{\text{act}2}, \ldots, f_{\text{act}|H|}$ mit H als Menge der versteckten Neurone durchzunummerieren – was die Erklärungen aber unübersichtlich macht. Insofern verwende ich einfach für alle Aktivierungsfunktionen die Bezeichnung f_{act} und betrachte σ und c als Größen, die zwar für einzelne Neurone definiert, aber nicht direkt in der Aktivierungsfunktion enthalten sind.

Dem Leser fällt bestimmt auf, dass die Gaußglocke bei ihren vielen Verwendungen in anderer Literatur oft mit einem multiplikativen Faktor versehen wird – aufgrund der sowieso vorhandenen Multiplikation durch die nachfolgenden Gewichte und der Aufwiegbarkeit von Konstantenmultiplikationen brauchen wir diesen Faktor jedoch hier nicht (zumal das Integral der Gaußglocke für unsere Zwecke nicht immer 1 sein darf) und lassen ihn daher einfach weg.

6.2.2 Einige analytische Gedanken im Vorfeld zum Training

Die Ausgabe y_Ω eines RBF-Ausgabeneurons Ω ergibt sich also, indem wir die Funktionen eines RBF-Neurons zusammensetzen zu

$$y_\Omega = \sum_{h \in H} w_{h,\Omega} \cdot f_{\text{act}}\left(||x - c_h||\right). \tag{6.4}$$

Nehmen wir an, wir besitzen wie beim Multilayerperceptron eine Menge P, die $|P|$ viele Trainingsbeispiele (p, t) enthält. Dann erhalten wir $|P|$ viele Funktionen der Form

$$y_\Omega = \sum_{h \in H} w_{h,\Omega} \cdot f_{\text{act}}\left(||p - c_h||\right), \tag{6.5}$$

nämlich für jedes Trainingsbeispiel p eine.

Das Ziel des Aufwands ist es natürlich wieder, die Ausgabe y für alle Trainingsmuster p gegen den zugehörigen Teaching Input t gehen zu lassen.

6.2.2.1 Gewichte können einfach als Lösung eines Gleichungssystems ausgerechnet werden

Wir sehen also, dass wir $|P|$ viele Gleichungen gegeben haben. Betrachten wir nun die Breiten $\sigma_1, \sigma_2, \ldots, \sigma_k$, die Zentren $c_1, c_2, \ldots, c_k$ und die Trainingsbeispiele p samt Teaching Input t als gegeben. *Gesucht* sind die Gewichte $w_{h,\Omega}$, wovon es $|H|$ Stück für ein einzelnes Ausgabeneuron Ω gibt. Wir können unser Problem also als Gleichungssystem sehen, da das einzige, was wir im Moment verändern möchten, die Gewichte sind.

Dies bringt uns dazu, eine Fallunterscheidung zu treffen bezüglich der Anzahl der Trainingsbeispiele $|P|$ und der Anzahl der RBF-Neurone $|H|$:

$|P| = |H|$: Ist die Zahl der RBF-Neurone gleich der Zahl der Muster, also $|P| = |H|$, so können wir die Gleichung auf eine Matrixmultiplikation

$$T = M \cdot G \tag{6.6}$$

$$\Leftrightarrow \quad M^{-1} \cdot T = M^{-1} \cdot M \cdot G \tag{6.7}$$

$$\Leftrightarrow \quad M^{-1} \cdot T = E \cdot G \tag{6.8}$$

$$\Leftrightarrow \quad M^{-1} \cdot T = G \tag{6.9}$$

zurückführen, wobei

- ▷ T der Vektor der Teaching Inputs für alle Trainingsbeispiele ist,
- ▷ M die $|P| \times |H|$-Matrix der Ausgaben von allen $|H|$ RBF-Neuronen zu $|P|$ vielen Beispielen (Wir erinnern uns: $|P| = |H|$, die Matrix ist quadratisch und daher können wir versuchen, sie zu invertieren),
- ▷ G der Vektor der gewünschten Gewichte und
- ▷ E eine Einheitsmatrix passend zu G.

Wir können also die Gewichte mathematisch gesehen einfach ausrechnen: Im Fall $|P| = |H|$ haben wir pro Trainingsbeispiel genau ein RBF-Neuron zur Verfügung. Das heißt nichts anderes, als dass das Netz nach erfolgtem Errechnen der Gewichte die $|P|$ vielen Stützstellen, die wir haben, exakt trifft, also eine ***exakte Interpolation*** durchführt – für ein solches Gleichungs-Ausrechnen benötigen wir aber natürlich kein RBF-Netz, so dass wir zum nächsten Fall übergehen wollen.

Exakte Interpolation ist nicht mit dem bei den MLPs erwähnten Auswendiglernen zu verwechseln: Erstens reden wir im Moment noch gar nicht über das Trainieren

von RBF-Netzen, zweitens kann es auch sehr gut für uns und durchaus gewollt sein, wenn das Netz zwischen den Stützstellen exakt interpoliert.

$|P| < |H|$: Das Gleichungssystem ist unterbestimmt, es existieren mehr RBF-Neurone als Trainingsbeispiele, also $|P| < |H|$. Dieser Fall taucht natürlich normalerweise nicht sehr oft auf. In diesem Fall gibt es eine Lösungsvielfalt, die wir gar nicht im Detail brauchen: Wir können uns aus vielen offensichtlich möglichen Sätzen von Gewichten einen auswählen.

$|P| > |H|$: Für die weitere Betrachtung am interessantesten ist jedoch der Fall, dass es signifikant mehr Trainingsbeispiele gibt als RBF-Neurone, also gilt $|P| > |H|$ – wir wollen also wieder die *Generalisierungsfähigkeit* der Neuronalen Netze nutzen.

Wenn wir viel mehr Trainingsmuster als RBF-Neurone haben, können wir fairerweise nicht mehr davon ausgehen, dass jedes Trainingsmuster exakt getroffen wird. Wenn wir die Punkte also nicht exakt treffen können und daher auch nicht nur *interpolieren* können wie im obigen Idealfall $|P| = |H|$, so müssen wir versuchen, eine Funktion zu finden, die unsere Trainingsmenge P so genau wie möglich ***approximiert***: Wir versuchen wie beim MLP, die Summe der Fehlerquadrate auf ein Minimum zu reduzieren.

Wie fahren wir also im Fall $|P| > |H|$ mit der Berechnung fort? Wie oben müssen wir, um das Gleichungssystem zu lösen, eine Matrixmultiplikation

$$T = M \cdot G \tag{6.10}$$

mit einer Matrix M lösen. Problem ist aber, dass wir die $|P| \times |H|$-Matrix M dieses mal nicht invertieren können, weil sie nicht quadratisch ist (es gilt $|P| \neq |H|$). Hier müssen wir also, um überhaupt weiterzukommen, die ***Moore-Penrose-Pseudoinverse*** mit

$$M^+ = (M^T \cdot M)^{-1} \cdot M^T \tag{6.11}$$

verwenden. Die Moore-Penrose-Pseudoinverse ist nicht die Inverse einer Matrix, kann aber in diesem Fall so verwendet werden[1]. Wir erhalten Gleichungen, die denen im Fall $|P| = |H|$ sehr ähnlich sind:

$$T = M \cdot G \tag{6.12}$$
$$\Leftrightarrow \quad M^+ \cdot T = M^+ \cdot M \cdot G \tag{6.13}$$
$$\Leftrightarrow \quad M^+ \cdot T = E \cdot G \tag{6.14}$$
$$\Leftrightarrow \quad M^+ \cdot T = G \tag{6.15}$$

Ein weiterer Hauptgrund für die Verwendung der Moore-Penrose-Pseudoinversen ist hier, dass sie die quadratische Abweichung minimiert (was unser Ziel ist): Die Schätzung des Vektors G in Gleichung 6.15 entspricht dem aus der Statistik bekannten ***Gauß-Markov-Modell*** zur Minimierung des quadratischen Fehlers. In den obigen Gleichungen 6.11 auf der linken Seite und folgenden sei das T in M^T (der *transponierten* Matrix M) bitte nicht mit dem T des Vektors aller Teaching Inputs zu verwechseln.

6.2.2.2 Die Verallgemeinerung auf mehrere Ausgaben ist trivial und wenig rechenintensiv

Wir haben also jeweils einen mathematisch exakten Weg gefunden, die Gewichte direkt zu errechnen. Was passiert nun bei mehreren Ausgabeneuronen, also $|O| > 1$, wobei O wie gewohnt die Menge der Ausgabeneurone Ω ist?

In diesem Fall ändert sich, wie wir schon angedeutet haben, nicht viel: Diese weiteren Ausgabeneurone haben ja einen eigenen Satz Gewichte, während wir die σ und c der RBF-Schicht nicht ändern. Für ein RBF-Netz ist es also für gegebene σ und c einfach, sehr viele Outputneurone zu realisieren, da wir nur für jedes neue Outputneuron Ω einzeln den Vektor der zu ihm führenden Gewichte

$$G_\Omega = M^+ \cdot T_\Omega \tag{6.16}$$

errechnen müssen, wobei die sehr aufwändig zu errechnende Matrix M^+ immer gleich bleibt: Das Hinzufügen von mehr Ausgabeneuronen ist also, was die Berechnungskomplexität angeht, recht preiswert.

1 Insbesondere gilt $M^+ = M^{-1}$, falls M invertierbar. Auf die Begründung für diese Umstände und Verwendungsmöglichkeiten von M^+ möchte ich hier nicht weiter eingehen – diese sind aber in der Linearen Algebra einfach zu finden.

6.2.2.3 Berechnungsaufwand und Genauigkeit

Bei realistischen Problemstellungen gilt jedoch in aller Regel, dass es *wesentlich* mehr Trainingsbeispiele als RBF-Neurone gibt, also $|P| \gg |H|$: Man kann ohne weiteres den Wunsch haben, mit 10^6 Trainingsbeispielen zu trainieren. Wir können zwar theoretisch mathematisch korrekt an der Tafel die Terme für die richtige Lösung finden (in sehr sehr langer Zeit), die Berechnung am Computer erweist sich aber oft als ungenau und sehr zeitaufwändig (Matrixinversionen haben einen großen Rechenaufwand).

Weiterhin ist unsere Moore-Penrose-Pseudoinverse, trotz numerischer Stabilität, noch keine Garantie dafür, dass der Outputvektor dem Teachingvektor entspricht, da bei den aufwändigen Berechnungen *sehr* viele Ungenauigkeiten auftreten können, obwohl die Rechenwege mathematisch natürlich korrekt sind: Unsere Computer können uns die pseudoinversen Matrizen auch nur *näherungsweise* liefern (wenn auch gute Näherungen). De facto erhalten wir also auch nur eine Näherung der richtigen Gewichte (womöglich mit vielen aufgeschaukelten Ungenauigkeiten) und damit auch nur eine (vielleicht sehr grobe oder gar unerkennbare) Näherung der Outputwerte an den gewünschten Output.

Falls wir genug Rechenpower besitzen, um einen Gewichtsvektor analytisch zu bestimmen, sollten wir ihn also auf jeden Fall nur als Startwert für unseren Lernvorgang benutzen, womit wir zu den wirklichen *Trainingsverfahren* kommen – aber sonst wäre es ja auch langweilig, oder?

6.3 Kombinationen aus Gleichungssystem und Gradientenverfahren sind zum Training sinnvoll

Analog zum MLP führen wir also zum Finden passender Gewichte einen Gradientenabstieg durch, und zwar über die bereits hinlänglich bekannte *Delta-Regel*. Backpropagation ist hier gar nicht notwendig, da wir nur eine einzige Gewichtsschicht trainieren müssen – ein Umstand, der sehr rechenzeitfreundlich ist.

Wie wir wissen, lautet die Delta-Regel an sich

$$\Delta w_{h,\Omega} = \eta \cdot \delta_\Omega \cdot o_h, \tag{6.17}$$

wobei wir in unserem Fall nun einsetzen:

$$\Delta w_{h,\Omega} = \eta \cdot (t_\Omega - y_\Omega) \cdot f_{\text{act}}(||p - c_h||) \tag{6.18}$$

Ich möchte noch einmal ausdrücklich darauf hinweisen, dass es sehr beliebt ist, das Training in zwei Phasen zu unterteilen, indem man zunächst einen Gewichtssatz analytisch berechnet und diesen mit der Delta-Regel nachtrainiert.

Oft wird das Training sogar in drei Phasen gegliedert: Es bleibt nämlich noch die Frage zu klären, ob man offline oder online lernt. Hier kann man Ähnliches sagen wie bei Multilayerperceptrons: Es wird oft zunächst online gelernt (schnellere Bewegung über die Fehleroberfläche). Anschließend, wenn man der Meinung ist, sich der Lösung zu nähern, werden in einer dritten Lernphase noch einmal die Fehler aufkumuliert und für eine noch genauere Annäherung offline gelernt. Ähnlich wie bei den MLPs erreicht man aber hier mit vielen Methoden gute Erfolge.

Wir haben aber schon angedeutet, dass man auch andere Dinge an einem RBF-Netz optimieren kann als nur die Gewichte vor der Outputschicht – betrachten wir also einmal die Möglichkeiten, die σ und c zu variieren.

6.3.1 Zentren und Breiten von RBF-Neuronen zu bestimmen, ist nicht immer trivial

Es ist klar, dass man die Approximationsgenauigkeit von RBF-Netzen erhöhen kann, indem man die Breiten und Positionen der Gaußglocken im Inputraum an das zu approximierende Problem anpasst. Es gibt mehrere Methoden, mit den Zentren c und Breiten σ der Gaußglocken zu verfahren:

Feste Wahl: Die Zentren und Breiten kann man fest und ohne Rücksicht auf die Muster wählen – hiervon sind wir bis jetzt ausgegangen.

Bedingte, feste Wahl: Wieder werden Zentren und Breiten fest gewählt, man besitzt allerdings Vorwissen über die zu approximierende Funktion und kann sich etwas danach richten.

Adaptiv zum Lernprozess: Zweifellos die eleganteste Variante, aber natürlich auch die anspruchsvollste. Eine Realisierung dieses Ansatzes wird in diesem Kapitel nicht besprochen, ist aber im Zusammenhang mit einer anderen Netztopologie zu finden (Abschnitt 10.6.1).

6.3.1.1 Feste Wahl

In jedem Fall ist es das Ziel, den Eingangsraum möglichst gleichmäßig abzudecken. Hier kann man dann Breiten von $\frac{2}{3}$ des Abstandes der Zentren zueinander wählen, so dass

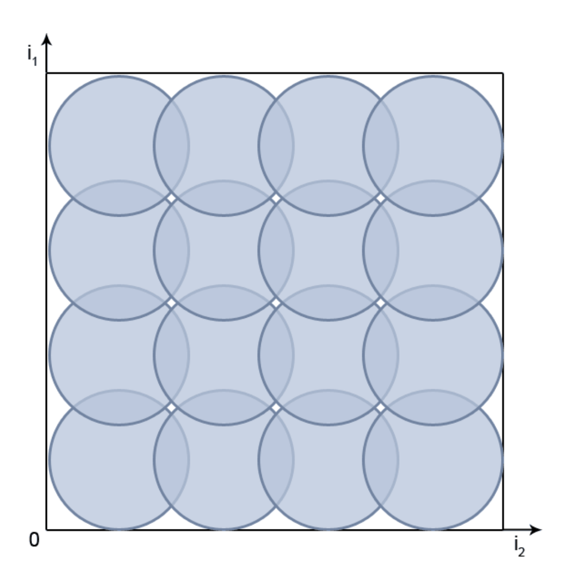


Abbildung 6.6: Beispiel einer gleichmäßigen Abdeckung eines zweidimensionalen Inputraumes durch Radialbasisfunktionen.

sich die Gaußglocken zu ca. „einem Drittel“[2] überlappen (Abb. 6.6). Je dichter wir die Glocken setzen, desto genauer, aber desto Rechenaufwändiger wird das Ganze.

Dies mag sehr unelegant aussehen, allerdings kommen wir in Bereichen der Funktionsapproximation kaum um die gleichmäßige Abdeckung herum – hier bringt es uns nämlich im Allgemeinen wenig, wenn die zu approximierende Funktion an einigen Stellen sehr genau repräsentiert wird, an anderen Stellen aber nur 0 als Rückgabewert geliefert wird.

Allerdings erfordert eine hohe Eingangsdimension gigantisch viele RBF-Neurone, was den Rechenaufwand exponentiell zur Dimension in die Höhe schnellen läßt – und dafür sorgt, dass wir sechs- bis zehndimensionale Probleme bei RBF-Netzen bereits „hochdimensional“ nennen (ein MLP macht hier beispielsweise überhaupt keine Probleme).

2 Es ist klar, dass eine Gaußglocke mathematisch unendlich breit ist, ich bitte also den Leser, diese saloppe Formulierung zu entschuldigen.

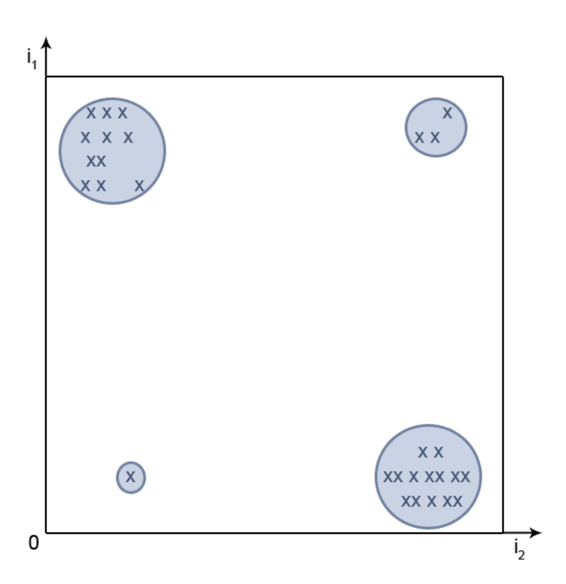


Abbildung 6.7: Beispiel einer ungleichmäßigen Abdeckung eines zweidimensionalen Inputraumes, über den wir Vorwissen besitzen, durch Radialbasisfunktionen.

6.3.1.2 Bedingte, feste Wahl

Angenommen, unsere Trainingsbeispiele sind nicht gleichmäßig über den Eingangsraum verteilt – dann liegt es nahe, die Zentren und Sigmas der RBF-Neurone anhand der Musterverteilung auszurichten. Man kann also seine Trainingsmuster mit statistischen Verfahren wie einer *Clusteranalyse* analysieren und so herausfinden, ob es statistische Gegebenheiten gibt, nach denen wir unsere Verteilung der Zentren und Sigmas richten sollten (Abb. 6.7).

Eine trivialere Möglichkeit wäre es, $|H|$ viele Zentren auf zufällig aus der Mustermenge ausgewählte Positionen zu setzen: Es bestünde also bei dieser Vorgehensweise für jedes Trainingsmuster p die Möglichkeit, direkt Zentrum eines Neurons zu sein (Abb. 6.8 auf der folgenden Seite). Das ist auch noch nicht sehr elegant, aber keine schlechte Lösung, wenn es schnell gehen soll. Bei dieser Vorgehensweise werden die Breiten in der Regel fest gewählt.

Wenn wir Grund zu der Annahme haben, dass die Menge der Trainingsbeispiele Häufungspunkte besitzt, können wir Clusteringverfahren benutzen, um diese zu finden. Es gibt verschiedene Arten, Cluster in einer beliebigdimensionalen Menge von Punkten zu finden, von denen wir einige im Exkurs A kennenlernen werden. Ein neuronales

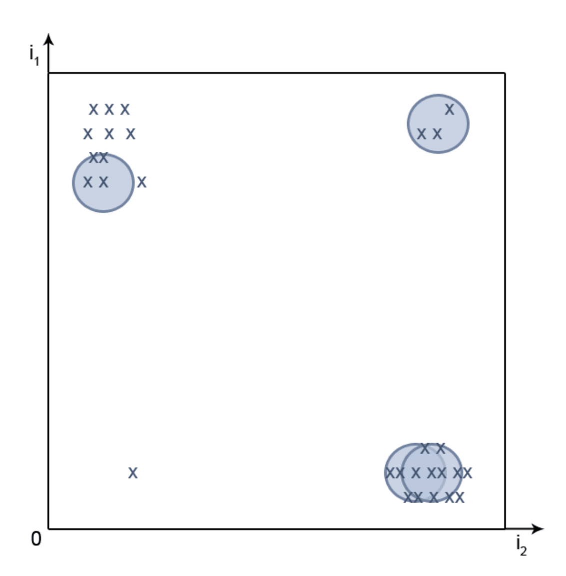


Abbildung 6.8: Beispiel einer ungleichmäßigen Abdeckung eines zweidimensionalen Inputraumes durch Radialbasisfunktionen. Die Breiten wurden fest gewählt, die Zentren der Neurone zufällig über die Trainingsmuster verteilt. Diese Verteilung kann natürlich leicht zu unrepräsentativen Ergebnissen führen, wie auch deutlich an dem allein stehenden Datenpunkt unten links zu sehen ist.

Clusteringverfahren sind die sogenannten ROLFs (Abschnitt A.5), und auch Self Organizing Maps haben sich als sinnvoll im Zusammenhang mit der Positionsbestimmung von RBF-Neuronen erwiesen (Abschnitt 10.6.1). Bei den ROLFs können sogar Anhaltspunkte für sinnvolle Radien der RBF-Neurone gefunden werden. Auch mit *Learning Vector Quantisation* (Kap. 9) wurden schon gute Ergebnisse erzielt. Alle diese Verfahren haben nichts direkt mit den RBF-Netzen zu tun, sondern dienen der reinen Erzeugung von Vorwissen – wir behandeln sie daher nicht in diesem Kapitel, sondern eigenständig in den genannten Kapiteln.

Ein weiterer Ansatz ist, auf Bewährtes zurückzugreifen: Wir können an den Positionen der Zentren drehen und schauen, wie sich unsere Fehlerfunktion Err damit verändert – einen Gradientenabstieg, wie von den MLPs gewohnt. Wir können auch auf gleiche Weise schauen, wie der Fehler von den Werten σ abhängt. Analog zur Herleitung von Backpropagation bilden wir also die Ableitungen

$$\frac{\partial \mathrm{Err}(\sigma_h c_h)}{\partial \sigma_h} \quad \text{und} \quad \frac{\partial \mathrm{Err}(\sigma_h c_h)}{\partial c_h}.$$

Da die Herleitung dieser Terme sich analog zu der von Backpropagation verhält, wollen wir sie hier nicht ausführen.

Die Erfahrung zeigt aber, dass es keine sehr überzeugenden Ergebnisse liefert, wenn wir betrachten, wie sich der Fehler abhängig von den Zentren und Sigmas verhält. Auch wenn die Mathematik uns lehrt, dass Verfahren dieser Art erfolgversprechend sind, ist der Gradientenabstieg, wie wir wissen, problembehaftet bei sehr zerklüfteten Fehleroberflächen.

Genau hier liegt der Knackpunkt: RBF-Netze bilden naturgemäß *sehr* zerklüftete Fehleroberflächen, denn wenn wir an einem c oder σ etwas verändern, verändern wir das Aussehen der Fehlerfunktion sehr stark.

6.4 Wachsende RBF-Netze passen die Neuronendichte automatisch an

Bei ***wachsenden RBF-Netzen*** ist die Anzahl $|H|$ der RBF-Neurone nicht konstant. Man wählt vorab eine bestimmte Zahl $|H|$ Neurone, sowie deren Zentren c_h und Breiten σ_h (beispielsweise anhand eines Clusteringverfahrens) und erweitert bzw. reduziert diese anschließend. Im Folgenden werden nur einfache Mechanismen angeschnitten, genauere Informationen finden Sie unter [Fri94].

6.4.1 Neurone werden Stellen großen Fehlers hinzugefügt

Nach Bildung dieser Startkonfiguration berechnet man analytisch den Vektor der Gewichte G. Man errechnet als nächstes alle spezifischen Fehler Err_p bezüglich der Menge P der Trainingsbeispiele und sucht den *maximalen* spezifischen Fehler

$$\max_P(\text{Err}_p).$$

Die Erweiterung des Netzes ist einfach: An die Stelle dieses größten Fehlers setzen wir nun ein neues RBF-Neuron. Hierbei müssen wir natürlich ein wenig aufpassen: Bei kleinen σ beeinflussen sich die Neurone nur bei wenig Entfernung zueinander. Bei großen σ hingegen findet aufgrund der Überlappung der Gaußglocken eine große Beeinflussung der schon vorhandenen Neurone durch das neue Neuron statt.

Es liegt also nahe, dass wir beim Hinzufügen eines neuen Neurons die bereits vorhandenen RBF-Neurone etwas anpassen.

Diese Anpassung geschieht salopp gesagt, indem man die Zentren c der anderen Neurone etwas von dem neuen Neuron wegbewegt und deren Breite σ etwas verkleinert. Danach wird der aktuelle Outputvektor y des Netzes mit dem Teaching Input t verglichen und der Gewichtsvektor G durch Training verbessert. Anschließend kann man, sofern erforderlich, wieder ein neues Neuron einfügen. Dieses Verfahren eignet sich insbesondere für Funktionsapproximationen.

6.4.2 Begrenzung der Neuronenanzahl

Es ist hierbei unbedingt darauf zu achten, dass das Netz nicht ins Unendliche wächst, was sehr schnell passieren kann. Es ist also sinnvoll, sich im Vorhinein eine Maximalanzahl für Neurone $|H|_{\max}$ zu definieren.

6.4.3 Weniger wichtige Neurone werden gelöscht

Dies bringt uns zu der Frage, ob man noch weiterlernen kann, wenn diese Grenze $|H|_{\max}$ erreicht ist. Auch hier ist dem Lernen noch kein Riegel vorgeschoben: Man sucht sich das „unwichtigste" Neuron und löscht es. Ein Neuron ist beispielsweise für das Netz nicht wichtig, wenn es ein anderes Neuron gibt, welches fast genau das gleiche tut: Es kommt oft vor, dass sich zwei Gaußglocken genau überlappen, an solchen Stellen würde beispielsweise ein einziges Neuron mit entsprechend größerer Höhe seiner Gaußglocke genügen.

Automatisierte Verfahren für das Finden von weniger relevanten Neuronen zu entwickeln, ist aber sehr stark problemabhängig und sei an dieser Stelle dem Programmierer überlassen.

Mit RBF-Netzen und Multilayerperceptrons haben wir nun bereits zwei Netzparadigmen für ähnliche Problemstellungen kennengelernt und ausführlich betrachtet. Wir wollen diese beiden Paradigmen daher einander gegenüberstellen und ihre Vor- und Nachteile vergleichen.

6.5 Gegenüberstellung von RBF-Netzen und Multilayerperceptrons

Wir nehmen den Vergleich von Multilayerperceptrons und RBF-Netzen anhand verschiedener Aspekte vor.

Eingabedimension: Bei RBF-Netzen ist in hochdimensionalen Funktionsräumen etwas Vorsicht geboten, da das Netz sehr schnell sehr speicher- und rechenaufwändig werden kann – hier macht ein Multilayerperceptron weniger Probleme, da dessen Neuronenanzahl nicht exponentiell mit der Eingabedimension wächst.

Wahl der Zentren: Allerdings ist die Wahl der Zentren c bei RBF-Netzen (trotz der hier vorgestellten Ansätze) nach wie vor ein großes Problem – bitte nutzen Sie bei deren Anwendung also wirklich *jedes* Vorwissen, das sie haben. Solche Probleme haben wir beim MLP nicht.

Ausgabedimension: Vorteil der RBF-Netze ist, dass es dem Training wenig macht, wenn das Netz eine hohe Output-Dimension aufweist – ein Lernverfahren wie Backpropagation bei einem MLP wird dabei sehr in die Länge gezogen.

Extrapolation: *Vorteil und Nachteil* von RBF-Netzen ist die mangelnde Extrapolationsfähigkeit: Ein RBF-Netz liefert weit weg von den Zentren der RBF-Schicht das Ergebnis 0. Dies ist gut und schlecht: Zum einen extrapoliert es eben nicht, es ist im Gegensatz zum MLP hierfür nicht verwertbar (wobei wir beim MLP nie wissen können, ob die extrapolierten Werte vernünftig sind, die Erfahrung zeigt aber, dass MLPs hier gutmütig sind). Zum anderen hat das Netz aber im Gegensatz zum MLP die Möglichkeit, uns durch diese 0 zu sagen „Ich weiss es nicht“, was sehr vom Vorteil sein kann.

Läsionstoleranz: Für den Output eines MLPs ist es nicht so wichtig, wenn irgendwo ein Gewicht oder Neuron fehlt, er wird insgesamt ein wenig schlechter werden. Fehlt ein Gewicht oder Neuron beim RBF-Netz, so sind weite Teile des Outputs so gut wie unbeeinflusst – eine Stelle des Outputs ist jedoch sehr betroffen, weil eben direkt eine Gaußglocke fehlt. Hier kann man also wählen zwischen starkem lokalem Fehler bei Läsion und schwachem, aber globalem Fehler.

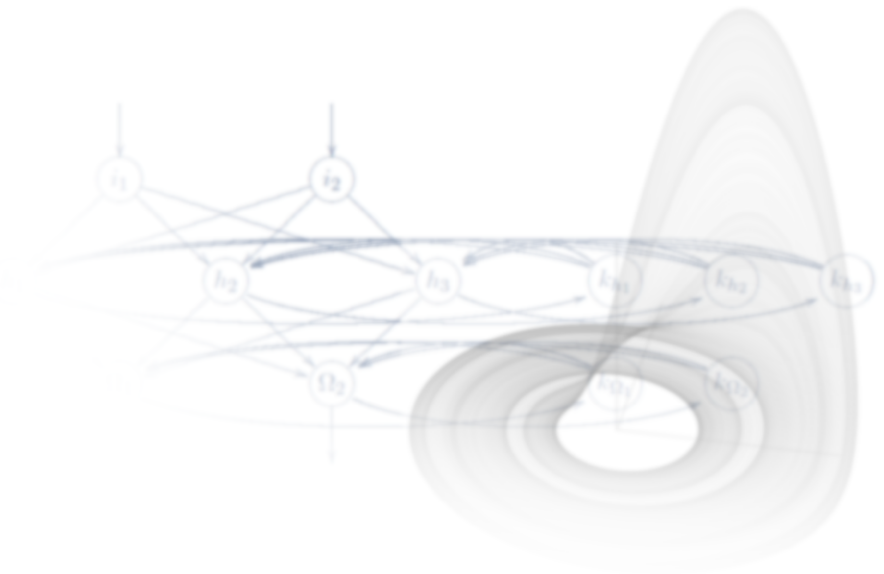
Verbreitung: Hier ist das MLP „im Vorteil“, da RBF-Netze wesentlich weniger angewandt werden – ein Umstand, der von professionellen Stellen nicht immer verstanden wird (was niedrigdimensionale Eingaberäume angeht). MLPs haben anscheinend eine wesentlich höhere Tradition und funktionieren zu gut, als dass es nötig wäre, sich ein paar Seiten in dieser Arbeit über RBF-Netze durchzulesen :-).

Übungsaufgaben

Aufgabe 14. Mit einem $|I|$-$|H|$-$|O|$-RBF-Netz mit festen Breiten und Zentren der Neurone soll eine Zielfunktion u approximiert werden. Hierfür sind $|P|$ viele Trainings-

beispiele der Form (p, t) der Funktion u gegeben. Es gelte $|P| > |H|$. Die Bestimmung der Gewichte soll analytisch über die *Moore-Penrose-Pseudoinverse* erfolgen. Geben Sie das Laufzeitverhalten bezüglich $|P|$ und $|O|$ möglichst genau an.

Hinweis: Für Matrixmultiplikationen und Matrixinvertierungen existieren Verfahren, welche effizienter sind als die kanonischen Verfahren. Für bessere Abschätzungen recherchieren Sie nach solchen Verfahren (und deren Komplexität). Geben Sie bei Ihrer Komplexitätsberechnung die verwendeten Verfahren samt ihrer Komplexität an.



Kapitel 7

Rückgekoppelte Netze

Gedanken über Netze, welche eigene interne Zustände besitzen.

Rückgekoppelte Netze im Allgemeinen bezeichnen Netze, die die Fähigkeit haben, sich durch ***Rückkopplungen*** selbst zu beeinflussen, beispielsweise indem die Ausgabe des Netzes in die folgenden Berechnungsschritte mit eingeht. Es gibt viele Arten rückgekoppelter Netze von nahezu beliebiger Gestalt, fast alle überschneidend als ***rückgekoppelte Neuronale Netze*** bezeichnet – demzufolge verwende ich für die wenigen hier vorgestellten Paradigmen den Namen ***rückgekoppelte Multilayerperceptrons***. W Dass man mit einem rückgekoppelten Netz mehr berechnen kann als mit einem normalen MLP, ist offensichtlich: Setzt man die Rückkopplungsgewichte auf 0, reduziert man das rückgekoppelte Netz ja auf ein normales MLP. Insbesondere erzeugt die Rückkopplung verschiedene netzinterne Zustände, so dass auch auf gleiche Eingaben im Kontext des Netzzustands verschiedene Ausgaben ausgegeben werden können.

Rückgekoppelte Netze an sich besitzen eine große Dynamik, die mathematisch sehr schwer zu erfassen und äußerst umfangreich zu betrachten ist. Das Ziel dieses Kapitels wird es nur sein, kurz zu betrachten, wie Rückkopplungen aufgebaut sein können und netzinterne Zustände erzeugt werden können. So werde ich nur kurz zwei Paradigmen rückgekoppelter Netze vorstellen, und anschließend grob deren Training umreißen.

Bei einem rückgekoppelten Netz können für eine zeitlich konstante Eingabe x verschiedene Dinge passieren: Zum einen kann das Netz konvergieren, sich also in einen festen Zustand bringen und irgendwann eine feste Ausgabe y ausgeben, oder es konvergiert eben nicht bzw. nach so langer Zeit, dass wir es nicht mehr mitbekommen, was eine ständige Veränderung von y zur Folge hat.

Falls es nicht konvergiert, kann man beispielsweise die Ausgabe auf ***Periodika*** oder ***Attraktoren*** (Abb. 7.1 auf der folgenden Seite) untersuchen – wir können hier die

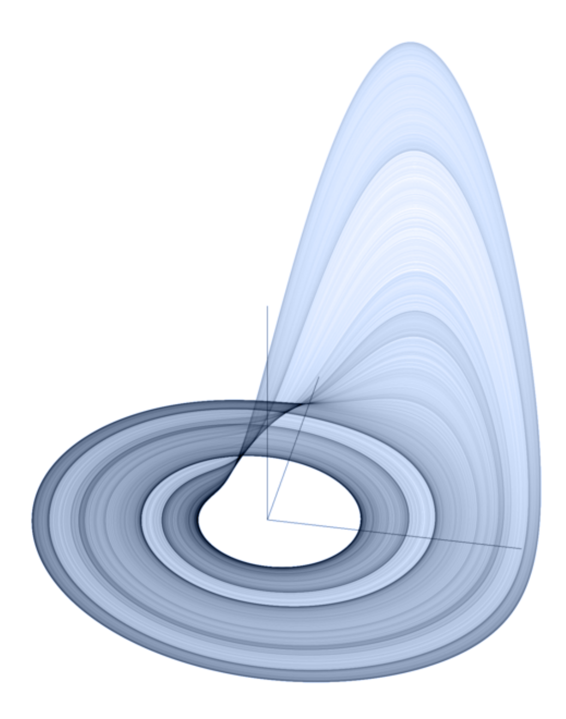


Abbildung 7.1: Der Rössler-Attraktor

komplette Vielfalt von ***dynamischen Systemen*** erwarten, weswegen ich auch speziell auf die Literatur zu dynamischen Systemen verweisen möchte.

Durch weitergehende Betrachtungen kann man dann herausfinden, was passiert, wenn bei rückgekoppelten Netzen der Input verändert wird.

Vorgestellt werden sollen in diesem Kapitel die verwandten Paradigmen rückgekoppelter Netze nach JORDAN und ELMAN.

7.1 Jordannetze

Ein ***Jordannetz*** [Jor86] ist ein Multilayerperceptron mit einer Menge K von sogenannten ***Kontextneuronen*** $k_1, k_2, \ldots, k_{|K|}$ – pro Output-Neuron existiert ein Kontextneuron (illustriert in Abb. 7.2 auf der rechten Seite). Ein Kontextneuron macht im Grunde nichts anderes, als einen Output zwischenzuspeichern, so dass er im nächsten Zeitschritt verwertet werden kann. Es gibt also gewichtete Verbindungen von jedem

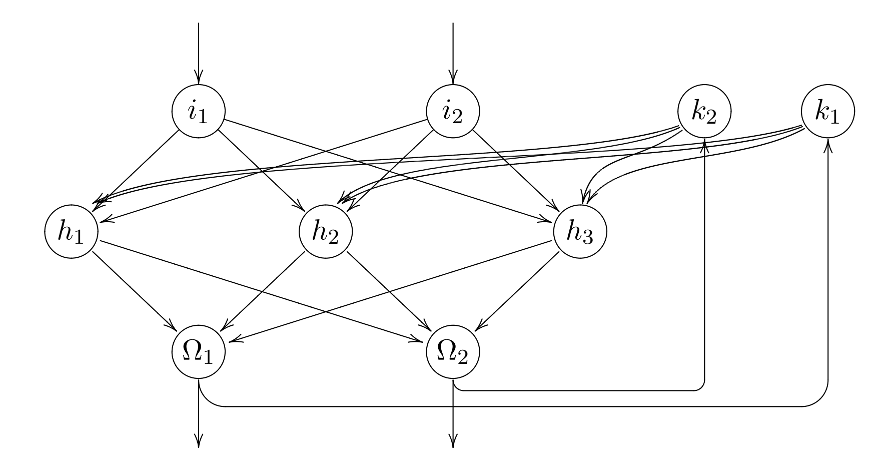


Abbildung 7.2: Darstellung eines Jordannetzes. Die Ausgabe des Netzes wird in den Kontextneuronen gepuffert und beim nächsten Zeitschritt zusammen mit der neuen Eingabe in das Netz eingebracht.

Ausgabeneuron zu einem Kontextneuron. Die gespeicherten Werte werden durch Vollverknüpfungen von den Kontextneuronen zur Eingabeschicht wieder an das eigentliche Netz zurückgegeben.

In der ursprünglichen Definition eines Jordannetzes sind die Kontextneurone auch noch über ein Verbindungsgewicht λ zu sich selbst rückgekoppelt – in den meisten Anwendungen wird diese Rückkopplung aber weggelassen, da das Jordannetz ohne diese zusätzlichen Rückkopplungen schon sehr dynamisch und schwer analysierbar ist.

Definition 7.1 (Kontextneuron)**.** Ein Kontextneuron k nimmt einen Outputwert eines anderen Neurons i zu einem Zeitpunkt t entgegen und gibt diesen im Zeitpunkt $(t + 1)$ wieder in das Netz ein.

Definition 7.2 (Jordannetz)**.** Ein Jordannetz ist ein Multilayerperceptron, welches pro Outputneuron ein Kontextneuron besitzt. Die Menge der Kontextneurone nennen wir K. Die Kontextneurone sind vollverknüpft in Richtung der Eingabeschicht des Netzes.

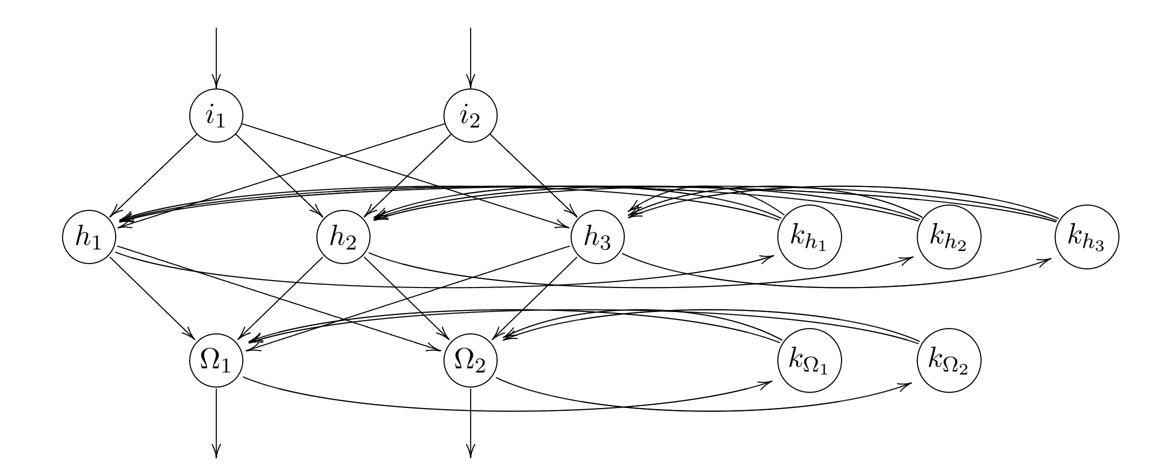


Abbildung 7.3: Darstellung eines Elmannetzes. Der ganze informationsverarbeitende Teil des Netzes ist sozusagen zweimal vorhanden. Die Ausgabe eines jeden Neurons (bis auf die der Eingabeneurone) wird gepuffert und in die zugeordnete Schicht wieder eingegeben. Ich habe die Kontextneurone zur Übersichtlichkeit anhand ihrer Vorbilder im eigentlichen Netz benannt, das muss man aber nicht so machen.

7.2 Elmannetze

Auch die ***Elmannetze*** (eine Variation der Jordannetze) [Elm90] haben Kontextneurone – allerdings pro informationsverarbeitender Neuronenschicht eine Schicht Kontextneurone (Abb. 7.3). Die Ausgaben eines jeden versteckten Neurons oder Ausgabeneurons werden also in die zugehörige Kontextschicht geleitet (wieder pro Neuron genau ein Kontextneuron) und von da im nächsten Zeitschritt wieder in die komplette Neuronenschicht eingegeben (auf dem Rückweg also wieder eine Vollverknüpfung). Es existiert also der gesamte informationsverarbeitende Teil[1] des MLPs noch einmal als „Kontextversion" – was die Dynamik und Zustandsvielfalt noch einmal wesentlich erhöht.

Gegenüber Jordannetzen haben Elmannetze oft den Vorteil, etwas zielgerichteter zu agieren, da jede Schicht auf ihren eigenen Kontext zurückgreifen kann.

1 Wir erinnern uns: die Inputschicht ist nicht informationsverarbeitend

Definition 7.3 (Elmannetz)**.** Ein Elmannetz ist ein MLP, welches pro informationsverarbeitendem Neuron ein Kontextneuron besitzt. Die Menge der Kontextneurone nennen wir K. Pro informationsverarbeitender Neuronenschicht existiert also eine Kontextschicht mit exakt gleichvielen Kontextneuronen. Jedes Neuron besitzt eine gewichtete Verbindung zu exakt einem Kontextneuron, während die Kontextschicht in Richtung ihrer Ursprungsschicht vollverknüpft ist.

Interessant ist nun, das Training rückgekoppelter Netze zu betrachten, da z.B. das normale Backpropagation of Error nicht auf rückgekoppelten Netzen funktionieren kann. Dieser Teil ist wieder etwas informeller gehalten, so dass ich dort keine formalen Definitionen verwenden werde.

7.3 Training rückgekoppelter Netze

Um das Training so anschaulich wie möglich darzustellen, müssen wir einige Vereinfachungen verabreden, die das Lernprinzip an sich aber nicht beeinträchtigen.

Gehen wir für das Training also davon aus, dass die Kontextneurone zu Anfang mit einer Eingabe initiiert werden, da sie sonst eine undefinierte Eingabe haben (das ist keine Vereinfachung, sondern wird tatsächlich gemacht).

Weiterhin verwenden wir für unsere Trainingsgehversuche ein Jordannetz, welches keine versteckte Neuronenschicht hat, so dass die Outputneurone direkt wieder Input liefern. Dies ist eine starke Vereinfachung, da in der Regel kompliziertere Netze verwendet werden. Dies verändert soweit aber nichts am Lernprinzip.

7.3.1 Unfolding in Time

Erinnern wir uns an unser eigentliches Lernverfahren für MLPs, *Backpropagation of Error*, das die Deltawerte rückwärts propagiert. Im Falle rückgekoppelter Netze würden die Deltawerte also zyklisch immer wieder rückwärts durch das Netz propagiert, was das Training schwieriger macht. Zum einen kann man nicht wissen, welche der vielen generierten Deltawerte für ein Gewicht nun für das Training gewählt werden sollen, also sinnvoll sind, zum anderen können wir auch nicht unbedingt wissen, wann das lernen nun beendet werden soll. Der Vorteil von rückgekoppelten Netzen ist eine große Zustandsdynamik innerhalb des Netzbetriebs – der Nachteil rückgekoppelter Netze ist, dass diese Dynamik auch dem Training zuteil wird und dieses somit erschwert.

Ein Lernansatz wäre der Versuch, die zeitlichen Zustände des Netzes aufzufalten (Abb. 7.4 auf der rechten Seite): Man löst die Rekursionen auf, indem man ein gleichförmiges Netz über die Kontextneurone setzt, also die Kontextneurone sozusagen die Ausgabeneurone des angefügten Netzes bilden. Allgemeiner ausgedrückt, Verfolgt man die Rekurrenzen zurück und hängt so „frühere" Instanzen von Neuronen in das Netz ein. So erzeugt man ein Großes, vorwärtsgerichtetes Netz, verliert aber im Gegenzug die Rekurrenzen und kann so das Netz mit Verfahren für nicht-rekurrente Netze trainieren In jede „Kopie" der Eingabeneurone wird hierbei die Eingabe als Teaching Input eingegeben. Dies kann man für eine diskrete Anzahl Zeitschritte tun. Wir nennen dieses Trainingsparadigma ***Unfolding in Time*** [MP69]. Nach der Auffaltung ist ein Training durch *Backpropagation of Error* möglich.

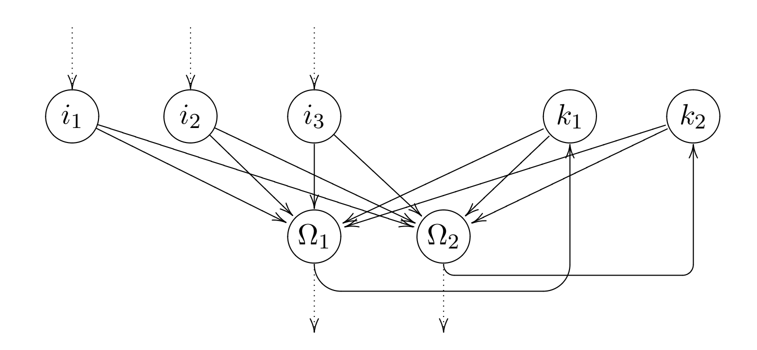
Offensichtlich erhält man für ein Gewicht $w_{i,j}$ aber mehrere Änderungswerte $\Delta w_{i,j}$, mit denen man auf verschiedene Weise verfahren kann: Aufkumulieren, Durchschnittsberechnung, etc. Kumuliert man sie einfach auf, können sich u.U. zu große Änderungen pro Gewicht ergeben, falls alle Änderungen das gleiche Vorzeichen haben. Demzufolge ist auch der Durchschnitt nicht zu verachten. Man könnte auch einen *discounting factor* einführen, der den Einfluss weiter in der Vergangenheit liegender $\Delta w_{i,j}$ abschwächt.

Unfolding in Time ist besonders dann sinnvoll, wenn man den Eindruck hat, dass die nähere Vergangenheit wichtiger für das Netz ist als die weiter entfernte, da Backpropagation in den von der Ausgabe weiter entfernten Schichten nur wenig Einfluss hat (wir erinnern uns: Der Einfluss von Backpropagation wird immer kleiner, je weiter man von der Ausgabeschicht weggeht).

Nachteile: Das Training des so auseinandergefalteten Netzes kann sehr lange dauern, da man unter Umständen eine große Anzahl Schichten produziert. Ein nicht mehr vernachlässigbares Problem ist die begrenzte Rechengenauigkeit normaler Computer, die bei derartig vielen geschachtelten Rechnungen sehr schnell erschöpft ist (der Einfluss von Backpropagation wird, je weiter man von den Ausgabeschichten weg kommt, immer kleiner, so dass man an diese Grenze stößt). Weiterhin kann das Verfahren bei mehreren Ebenen Kontextneurone sehr große zu trainierende Netze produzieren.

7.3.2 Teacher Forcing

Weitere Verfahren sind die deckungsgleichen ***Teacher Forcing*** und ***Open Loop Learning***. Sie schneiden während des Lernens die Rückkopplung auf: Während des Lernens tun wir einfach so, als gäbe es die Rückkopplung nicht, und legen den Teaching Input



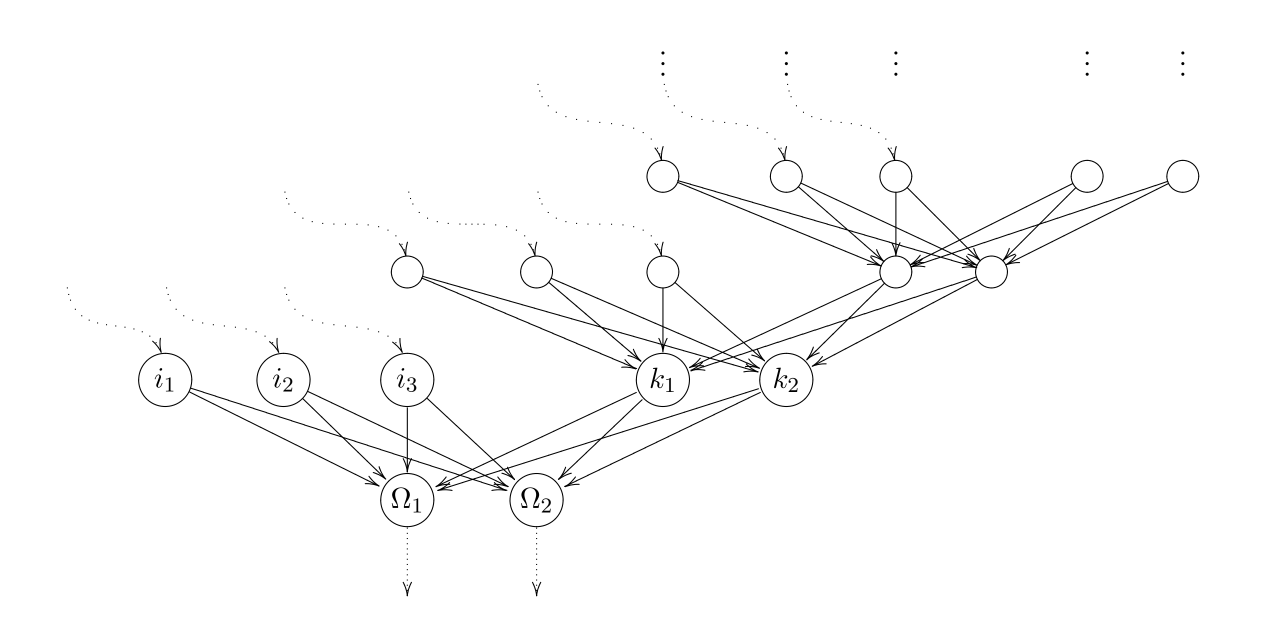


Abbildung 7.4: Darstellung des Unfolding in Time bei einem kleinen beispielhaften rückgekoppelten MLP. **Oben:** Das rückgekoppelte MLP. **Unten:** Das aufgefaltete Netz. Um der Übersichtlichkeit zu dienen, habe ich im aufgefalteten Netz nur dem untersten Teilnetz Benennungen hinzugefügt. Gepunktete Pfeile, welche in das Netz hineinführen, markieren Eingaben. Gepunktete Pfeile, welche aus dem Netz herausführen, markieren Ausgaben. Jede „Netzkopie" repräsentiert einen Zeitschritt des Netzes, unten ist der aktuellste Zeitschritt.

während des Trainings an den Kontextneuronen an. Auch so wird ein Backpropagation-Training möglich. Nachteil: Ein Teaching Input bei Kontextneuronen oder allgemein nicht-ausgabe-Neuronen ist ja nicht verfügbar.

7.3.3 Rekurrentes Backpropagation

Ein weiteres beliebtes Verfahren ohne beschränkten Zeithorizont ist ***rekurrentes Backpropagation***, das dem Problem mit Hilfe von Lösungsmethoden für Differentialgleichungen zu Leibe rückt [Pin87].

7.3.4 Training mit Evolution

Gerade bei rückgekoppelten Netzen haben sich aufgrund des ohnehin langen Trainings ***evolutionäre Algorithmen*** bewährt, da diese nicht nur in Bezug auf Rückkopplungen keinen Beschränkungen unterliegen, sondern bei geeigneter Wahl der Mutationsmechanismen auch weitere Vorteile haben: So können zum Beispiel Neurone und Gewichte angepasst und so die Netztopologie optimiert werden (es muss dann nach dem Lernen natürlich kein Jordan- oder Elmannetz herauskommen). Bei normalen MLPs hingegen sind die Evolutionsstrategien weniger gefragt, da sie natürlich viel mehr Zeit benötigen als ein gerichtetes Lernverfahren wie Backpropagation.



Kapitel 8

Hopfieldnetze

In einem magnetischen Feld übt jedes Teilchen Kraft auf jedes andere Teilchen aus, so dass sich die Teilchen insgesamt so ausrichten, wie es am energetisch günstigsten für sie ist. Wir kopieren diesen Mechanismus der Natur, um verrauschte Eingaben zu ihren richtigen Vorbildern zu korrigieren.

Ein weiteres überwacht lernendes Beispiel aus dem Zoo der Neuronalen Netze wurde von John Hopfield entwickelt: die nach ihm benannten ***Hopfieldnetze*** [Hop82]. Hopfield und seine physikalisch motivierten Netze haben viel zur Renaissance der Neuronalen Netze beigetragen.

8.1 Hopfieldnetze sind inspiriert durch Teilchen in einem magnetischen Feld

Die Idee für die Hopfieldnetze ist aus dem Verhalten von Teilchen im Magnetismus entstanden: Jedes Teilchen „redet“ (durch die magnetischen Kräfte) mit jedem anderen (also eine Vollverknüpfung), wobei es aber jeweils versucht, einen energetisch günstigen Zustand (sozusagen ein *Minimum der Energiefunktion*) zu erreichen. Diesen Eigenzustand kennen wir bei den Neuronen als Aktivierung. Die Teilchen bzw. Neurone drehen sich also alle und animieren sich dadurch wieder gegenseitig zur Drehung. Unser Neuronales Netz ist also sozusagen eine Wolke von Teilchen.

Ausgehend von der Tatsache, dass die Teilchen die Minima in der Energiefunktion selbsttätig aufspüren, hatte Hopfield nun die Idee, den „Drehwinkel“ der Teilchen zu nutzen, um Informationsverarbeitung zu betreiben: Warum nicht die Teilchen auf selbstdefinierten Funktionen Minima suchen lassen? Selbst wenn wir nur zwei dieser

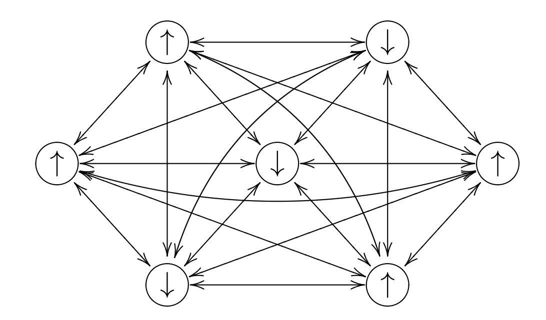


Abbildung 8.1: Darstellung eines beispielhaften Hopfieldnetzes. Die Pfeile ↑ und ↓ markieren die binären „Drehwinkel“. Durch die vollständige Verknüpfung der Neurone können keine Schichten voneinander abgegrenzt werden, so dass ein Hopfieldnetz einfach eine Menge von Neuronen umfasst.

Drehwinkel (***Spins***) verwenden, also eine *binäre Aktivierung*, werden wir feststellen, dass das entwickelte Hopfieldnetz erstaunliche Dynamik besitzt.

8.2 In einem Hopfieldnetz beeinflussen sich alle Neurone symmetrisch gegenseitig

Kurz gesagt besteht ein Hopfieldnetz also aus einer Menge K von untereinander vollverknüpften Neuronen mit binärer Aktivierung (wir verwenden ja nur zwei Drehwinkel), wobei die Gewichte zwischen den einzelnen Neuronen symmetrisch sind und ein Neuron *keine Direktverbindung* zu sich selbst aufweist (Abb. 8.1). Der *Zustand* von $|K|$ vielen Neuronen mit zwei möglichen Zuständen $\in \{-1, 1\}$ lässt sich also durch eine Zeichenkette $x \in \{-1, 1\}^{|K|}$ beschreiben.

Die Vollverknüpfung sorgt dafür, dass wir eine volle quadratische Matrix an Gewichten unter den Neuronen haben. Was die Gewichte bedeuten, wollen wir gleich erforschen. Weiter werden wir gleich durchschauen, nach welchen Regeln die Neurone sich drehen, also ihren Zustand ändern.

Die Vollverknüpfung sorgt weiterhin dafür, dass wir keine Input-, Output- oder versteckten Neurone kennen. Wir müssen uns also Gedanken machen, wie wir etwas in die $|K|$ Neurone eingeben.

Definition 8.1 (Hopfieldnetz)**.** Ein Hopfieldnetz besteht aus einer Menge K von vollverknüpften Neuronen ohne direkte Rückkopplungen. Die Aktivierungsfunktion der Neurone ist die binäre Schwellenwertfunktion mit Ausgaben $\in \{1, -1\}$.

Definition 8.2 (Zustand eines Hopfieldnetzes)**.** Die Gesamtheit der Aktivierungszustände aller Neurone ist der Zustand des Netzes. Der Netzzustand kann also als Binärstring $z \in \{-1, 1\}^{|K|}$ aufgefasst werden.

8.2.1 Eingabe und Ausgabe eines Hopfieldnetzes werden durch Neuronenzustände repräsentiert

Wir haben gelernt, dass das Netz, also die Menge der $|K|$ vielen Teilchen, von einem Zustand aus selbsttätig ein Minimum sucht. Ein Inputmuster eines Hopfieldnetzes ist genau so ein Zustand: Ein Binärstring $x \in \{-1, 1\}^{|K|}$, welcher die Neurone initialisiert. Das Netz sucht dann zur Eingabe das Minimum auf seiner Energieoberfläche (das wir vorher durch Eingabe von Trainingsbeispielen selbst definiert haben).

Woher wissen wir nun, dass das Minimum gefunden ist? Auch das ist einfach: Wenn das Netz stillsteht. Man kann beweisen, dass ein Hopfieldnetz mit symmetrischer Gewichtsmatrix und Nullen in der Diagonale *immer konvergiert* [CG88] , es wird also irgendwann still stehen. Die Ausgabe ist dann ein Binärstring $y \in \{-1, 1\}^{|K|}$, nämlich die Zustandszeichenkette des Netzes, das ein Minimum gefunden hat.

Betrachten wir nun die Inhalte der Gewichtsmatrix und die Regeln für den Zustandswechsel der Neurone etwas genauer.

Definition 8.3 (Eingabe und Ausgabe eines Hopfieldnetzes)**.** Die Eingabe in ein Hopfieldnetz ist ein Binärstring $x \in \{-1, 1\}^{|K|}$, welcher den Netzzustand initialisiert. Nach erfolgter Konvergenz des Netzes ist der aus dem neuen Netzzustand erzeugte Binärstring $y \in \{-1, 1\}^{|K|}$ die Ausgabe.

8.2.2 Bedeutung der Gewichte

Wir haben gesagt, dass die Neurone ihre Zustände, also ihre Ausrichtung von -1 nach 1 oder umgekehrt ändern. Diese Drehungen finden abhängig von den aktuellen Zuständen der anderen Neurone und von den Gewichten zu diesen statt. Die Gewichte sind also in der Lage, die Gesamtveränderung des Netzes zu steuern. Die Gewichte können positiv, negativ oder 0 sein. Hierbei gilt umgangsprachlich für ein Gewicht $w_{i,j}$ zwischen zwei Neuronen i und j:

Ist $w_{i,j}$ positiv, versucht es die beiden Neurone zur Gleichheit zu zwingen, je größer $w_{i,j}$, desto stärker ist der Zwang. Wenn das Neuron i den Zustand 1 hat, das Neuron j aber den Zustand -1, vermittelt ein hohes positives Gewicht den beiden Neuronen, dass es energetisch günstiger ist, wenn sie gleich sind.

Ist $w_{i,j}$ negativ, verhält es sich analog, nur werden hier i und j zur Unterschiedlichkeit gedrängt. Ein Neuron i mit Zustand -1 würde versuchen, ein Neuron j in den Zustand 1 zu drängen.

Null-Gewichte sorgen dafür, dass sich die beiden beteiligten Neurone nicht beeinflussen.

Die Gesamtheit der Gewichte beschreibt also offensichtlich den Weg zum nächsten Minimum der Energiefunktion vom aktuellen Netzzustand aus – wir wollen nun untersuchen, auf welche Weise die Neurone diesen Weg einschlagen.

8.2.3 Ein Neuron wechselt den Zustand anhand des Einflusses der anderen Neurone

Die Funktionsweise des einmal trainierten und mit einem Anfangszustand initialisierten Netzes liegt darin, die Zustände x_k der einzelnen Neurone k nach dem Schema

$$x_k(t) = f_{\text{act}}\left(\sum_{j\in K} w_{j,k} \cdot x_j(t-1)\right) \tag{8.1}$$

mit jedem Zeitschritt zu ändern, wobei die Funktion f_{act} in aller Regel die binäre Schwellenwert-Funktion (Abb. 8.2 auf der rechten Seite) mit Schwellenwert 0 ist. Umgangssprachlich: Ein Neuron k berechnet die Summe der $w_{j,k} \cdot x_j(t-1)$, die angibt, wie stark und in welche Richtung das Neuron k von den anderen Neuronen j gedrängt wird. Der neue Zustand des Netzes (Zeitpunkt t) ergibt sich also aus dem Netzzustand zum vorherigen Zeitpunkt $t-1$. Die Summe ergibt dann die Gesamtrichtung, in die das Neuron k gedrängt wird. Je nach Vorzeichen der Summe nimmt das Neuron den Zustand 1 oder -1 an.

Ein weiterer Unterschied der Hopfieldnetze zu anderen Netztopologien, welche wir bereits kennengelernt haben, ist das *asynchrone Update*: Es wird jedes mal ein Neuron k zufällig gewählt, welches dann die Aktivierung neu errechnet. Die neuen Aktivierungen der jeweils vorher geänderten Neurone nehmen also direkt Einfluss, ein Zeitschritt bezeichnet also die Änderung eines einzigen Neurons.

Ungeachtet der hier beschriebenen zufälligen Wahl des Neurons findet die Implementierung eines Hopfieldnetzes oft einfacher statt: die Neurone werden einfach nacheinander

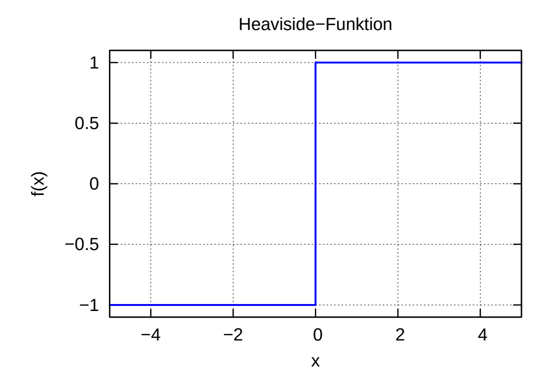


Abbildung 8.2: Darstellung der binären Schwellenwertfunktion.

durchgegangen und deren Aktivierungen neu berechnet – so lange, bis sich nichts mehr ändert.

Definition 8.4 (Zustandswechsel eines Hopfieldnetzes)**.** Der Zustandswechsel der Neurone findet asynchron statt, wobei das zu aktualisierende Neuron jeweils zufällig bestimmt wird und der neue Zustand durch die Vorschrift

$$x_k(t) = f_{\text{act}}\left(\sum_{j\in K} w_{j,k} \cdot x_j(t-1)\right)$$

gebildet wird.

Nachdem wir jetzt wissen, wie die Gewichte die Zustandsänderungen der Neurone beeinflussen und das gesamte Netz in Richtung eines Minimums treiben, ist nun noch die Frage offen, wie man den Gewichten beibringt, das Netz in Richtung *eines bestimmten* Minimums zu treiben.

8.3 Die Gewichtsmatrix wird direkt anhand der Trainingsbeispiele erzeugt

Es ist das Ziel, Minima auf der genannten Energieoberfläche zu erzeugen, in die das Netz bei einer Eingabe konvergiert. Wie schon bei vielen anderen Netzparadigmen, verwenden wir hier wieder eine Menge P von Mustereingaben $p \in \{1, -1\}^{|K|}$, die die Minima unserer Energieoberfläche darstellen.

Im Unterschied zu vielen anderen Netzparadigmen suchen wir hier nicht die Minima einer uns unbekannten Fehlerfunktion, sondern definieren uns selbst Minima auf einer solchen – mit dem Zweck, dass das Netz das naheliegendste Minimum von selbst annehmen soll, wenn man ihm eine Eingabe präsentiert. Dies ist für uns erst einmal ungewohnt – den Sinn und Zweck werden wir aber noch verstehen.

Das Training eines Hopfieldnetzes spielt sich grob gesagt so ab, dass man jedes Trainingsmuster *genau einmal* mit der im Folgenden beschriebenen Vorschrift trainiert (***Single Shot Learning***), wobei p_i und p_j die Zustände der Neurone i und j in dem Beispiel $p \in P$ sind:

$$w_{i,j} = \sum_{p \in P} p_i \cdot p_j \tag{8.2}$$

Hieraus ergibt sich dann die Gewichtsmatrix W. Umgangssprachlich: Wir initialisieren das Netz mit einem Trainingsmuster, und gehen dann alle Gewichte $w_{i,j}$ durch. Für jedes dieser Gewichte schauen wir: Sind die Neurone i, j im gleichen Zustand oder sind die Zustände unterschiedlich? Im ersten Fall addieren wir 1 zum Gewicht, im zweiten Fall -1.

Dies machen wir für alle Trainingsmuster $p \in P$. Zum Schluss haben also Gewichte $w_{i,j}$ hohe Werte, wenn i und j bei vielen Trainingsmustern übereingestimmt haben. Der hohe Wert sagt diesen Neuronen umgangssprachlich: „Es ist sehr oft energetisch günstig, wenn ihr den gleichen Zustand innehabt“. Entsprechendes gilt für negative Gewichte.

Durch dieses Training können wir also eine gewisse feste Anzahl Muster p in der Gewichtsmatrix abspeichern. Das Netz wird dann bei einer Eingabe x zu dem abgespeicherten Muster konvergieren, dass der Eingabe p am nächsten liegt.

Leider ist die Zahl der maximal speicherbaren und rekonstruierbaren Muster p auf

$$|P|_{\text{MAX}} \approx 0.139 \cdot |K| \tag{8.3}$$

beschränkt, was auch wiederum nur für orthogonale Muster gilt. Dies wurde durch genaue (und aufwändige) mathematische Analysen gezeigt, auf die wir jetzt nicht eingehen wollen. Gibt man mehr Muster als diese Anzahl ein, zerstört man bereits gespeicherte Informationen.

Definition 8.5 (Lernregel für Hopfieldnetze)**.** Die einzelnen Elemente der Gewichtsmatrix W werden durch das einmalige Ausführen der Lernregel

$$w_{i,j} = \sum_{p \in P} p_i \cdot p_j$$

bestimmt, wobei die Diagonale der Matrix mit Nullen belegt ist. Hierbei können nicht mehr als $|P|_{\text{MAX}} \approx 0.139 \cdot |K|$ Trainingsbeispiele funktionserhaltend trainiert werden.

Wir haben nun die Funktionalität von Hopfieldnetzen kennen gelernt, jedoch noch nicht ihren praktischen Nährwert.

8.4 Autoassoziation und traditionelle Anwendung

Hopfieldnetze, wie sie oben beschrieben wurden, sind sogenannte ***Autoassoziatoren***. Ein Autoassoziator a legt genau das oben beschriebene Verhalten an den Tag: Erstens gibt er bei Eingabe eines bekannten Musters p genau dieses bekannte Muster wieder aus, es gilt also

$$a(p) = p,$$

wobei a die Assoziator-Abbildung ist. Zum Zweiten, und genau hier liegt der Nährwert des Ganzen, funktioniert das auch mit Eingaben, die in der Nähe von einem Muster liegen:

$$a(p + \varepsilon) = p.$$

Der Autoassoziator ist hinterher in jedem Fall in einem stabilen Zustand, nämlich im Zustand p.

Nimmt man als Mustermenge P beispielsweise Buchstaben oder sonstige Schriftzeichen in Pixelform, so wird das Netz in der Lage sein, deformierte oder verrauschte Buchstaben mit hoher Wahrscheinlichkeit richtig zu erkennen (Abb. 8.3 auf der folgenden Seite).

Anwendung von Hopfieldnetzen sind daher prinzipiell ***Mustererkennung*** und Mustervervollständigung, so zum Beispiel Ende der 1980er Jahre die Erkennung von Postleitzahlen auf Briefen. Bald sind die Hopfieldnetze aber in den meisten ihrer Anwendungsgebiete von anderen Systemen überholt worden, so in der Buchstabenerkennung von modernen OCR-Systemen. Heute werden Hopfieldnetze so gut wie überhaupt nicht mehr verwendet, sie haben sich nicht durchgesetzt.

8.5 Heteroassoziation und Analogien zur neuronalen Datenspeicherung

Bis jetzt haben wir Hopfieldnetze kennengelernt, die für eine beliebige Eingabe in das naheliegendste Minimum einer statischen Energieoberfläche konvergieren.

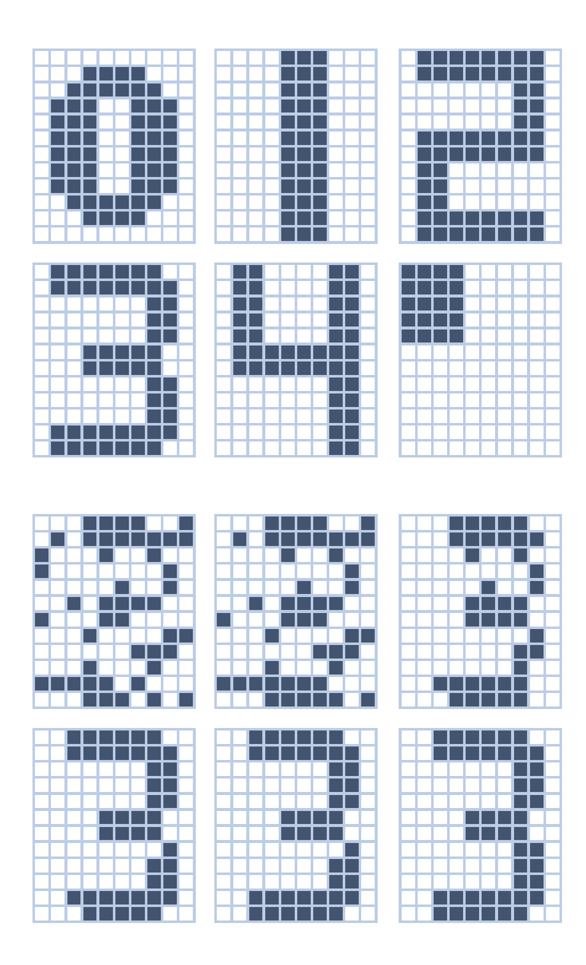


Abbildung 8.3: Darstellung der Konvergenz eines beispielhaften Hopfieldnetzes. Jedes der Bilder hat $10 \times 12 = 120$ binäre Pixel. Jeder Pixel entspricht im Hopfieldnetz einem Neuron. Oben sind die Trainingsbeispiele abgebildet, unten die Konvergenz einer stark verrauschten 3 zum korrespondierenden Trainingsbeispiel.

Eine weitere Variante wäre eine dynamische Energieoberfläche: Hier sieht die Energieoberfläche je nach aktuellem Zustand anders aus und wir erhalten keinen Autoassoziator mehr, sondern einen ***Heteroassoziator***. Für einen Heteroassoziator gilt nicht mehr

$$a(p + \varepsilon) = p,$$

sondern vielmehr

$$h(p + \varepsilon) = q,$$

was bedeutet, dass ein Muster auf ein anderes abgebildet wird. h ist die Heteroassoziator-Abbildung. Man erreicht solche Heteroassoziationen durch eine asymmetrische Gewichtsmatrix V.

Durch hintereinandergeschaltete Heteroassoziationen der Form

$$\begin{aligned} h(p + \varepsilon) &= q \\ h(q + \varepsilon) &= r \\ h(r + \varepsilon) &= s \\ &\vdots \\ h(z + \varepsilon) &= p \end{aligned}$$

wird es möglich, einen schnellen Zustandsdurchlauf

$$p \to q \to r \to s \to \ldots \to z \to p$$

zu provozieren, wobei ein einzelnes Muster aber niemals vollständig angenommen wird: Bevor ein Muster vollständig zustandegekommen ist, versucht die Heteroassoziation ja bereits, dessen Nachfolger zu erzeugen. Außerdem würde unser Netz nie zum Stillstand kommen, da es ja nach Erreichen des letzten Zustands z wieder zum ersten Zustand p übergeht.

8.5.1 Erzeugung der Heteroassoziationsmatrix

Wir erzeugen die Matrix V mit Elementen v sehr ähnlich der Autoassoziationsmatrix, wobei (pro Übergang) p das Trainingsbeispiel vor dem Übergang ist und q das aus p zu erzeugende Trainingsbeispiel:

$$v_{i,j} = \sum_{p,q \in P, p \neq q} p_i q_j \tag{8.4}$$

Die Diagonale der Matrix ist wieder mit Nullen belegt. Die Adaption der Neuronenzustände erfolgt im laufenden Betrieb, wie gehabt. Mehrere Übergänge können einfach durch Aufsummierung in die Matrix eingebracht werden, wobei auch hier wieder die genannte Begrenzung gegeben ist.

Definition 8.6 (Lernregel für Heteroassoziationsmatrix)**.** Für zwei Trainingsbeispiele p als Vorgänger und q als Nachfolger eines Heteroassoziationsübergangs ergeben sich die Gewichte der Heteroassoziationsmatrix V durch die Lernregel

$$v_{i,j} = \sum_{p,q \in P, p \neq q} p_i q_j,$$

wobei sich mehrere Heteroassoziationen durch einfache Aufsummierung in ein Netz einbringen lassen.

8.5.2 Stabilisierung der Heteroassoziationen

Wir haben oben das Problem angesprochen, dass die Muster nicht vollständig erzeugt werden, sondern schon vor Ende der Erzeugung bereits das nächste Muster in Angriff genommen wird.

Dieses Problem kann umgangen werden, indem das Netz nicht nur durch die Heteroassoziationsmatrix V beeinflusst wird, sondern zusätzlich durch die bereits bekannte Autoassoziationsmatrix W.

Zusätzlich wird die Neuronenadaptionsregel so verändert, dass konkurrierende Terme entstehen: Einer, der ein vorhandenes Muster autoassoziiert, und einer, der versucht, eben dieses Muster in seinen Nachfolger umzuwandeln. Die Assoziationsregel bewirkt hierbei, dass das Netz ein Muster stabilisiert, dort eine Weile bleibt, zum nächsten Muster übergeht, und so weiter.

$$x_i(t+1) = \\ f_{\text{act}}\left(\underbrace{\sum_{j \in K} w_{i,j} x_j(t)}_{\text{Autoassoziation}} + \underbrace{\sum_{k \in K} v_{i,k} x_k(t - \Delta t)}_{\text{Heteroassoziation}} \right) \tag{8.5}$$

Der Wert Δt bewirkt hierbei anschaulich gesprochen, dass der Einfluss der Matrix V verzögert eintritt, da sie sich nur auf eine um Δt zurückliegende Version des Netzes bezieht. Hierdurch ergibt sich ein Zustandswechsel, in dem die einzelnen Zustände aber

zwischendurch für kurze Zeit stabil sind. Setzen wir Δt auf z.B. zwanzig Schritte, so bekommt die asymmetrische Gewichtsmatrix jegliche Veränderung im Netz erst zwanzig Schritte später mit, so dass sie zunächst mit der Autoassoziationsmatrix zusammenarbeitet (da sie noch das Vorgängermuster vom aktuellen wahrnimmt) und erst später gegen sie.

8.5.3 Biologische Motivation der Heteroassoziation

Die Übergänge von stabilen in andere stabile Zustände sind hierbei biologisch stark motiviert: Es gab zumindest Anfang der 1990er Jahre Vermutungen, dass mit dem Hopfieldmodell eine Näherung der Zustandsdynamik im Gehirn erreicht wird, welches vieles durch Zustandsketten realisiert: Wenn ich Sie, lieber Leser, nun bitte, das Alphabet aufzusagen, werden Sie das in der Regel wesentlich besser schaffen als mir (bitte sofort versuchen) die Frage zu beantworten:

> *Welcher Buchstabe folgt im Alphabet auf den Buchstaben P?*

Ein anderes Beispiel ist das Phänomen, dass man etwas vergisst, jedoch den Ort noch kennt, an dem man zuletzt daran gedacht hat. Geht man nun an diesen Ort zurück, fällt einem das Vergessene oftmals wieder ein.

8.6 Kontinuierliche Hopfieldnetze

Bis jetzt haben wir nur Hopfieldnetze mit binären Aktivierungen erforscht. Hopfield beschrieb aber auch eine Version seiner Netze mit kontinuierlichen Aktivierungen [Hop84], die wir zumindest kurz anschauen wollen: ***kontinuierliche Hopfieldnetze***. Hier wird die Aktivierung nicht mehr durch die binäre Schwellenwertfunktion berechnet, sondern durch die Fermifunktion mit Temperaturparameter (Abb. 8.4 auf der folgenden Seite).

Auch hier ist das Netz für symmetrische Gewichtsmatrizen mit Nullen auf der Diagonalen stabil.

Hopfield selbst nennt als Anwendungsbeispiel für kontinuierliche Hopfieldnetze, recht gute Lösungen für das NP-harte Travelling Salesman Problem zu finden [HT85]. Nach einem in [Zel94] beschriebenen Feldversuch kann dieses Statement aber nicht ohne weiteres aufrecht erhalten werden. Es gibt heute aber ohnehin schnellere Algorithmen, um gute Lösungen für dieses Problem zu finden, weswegen das Hopfieldnetz auch hier keine Anwendung mehr finden kann.

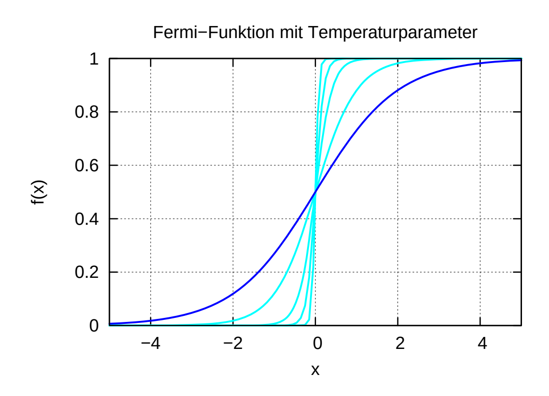


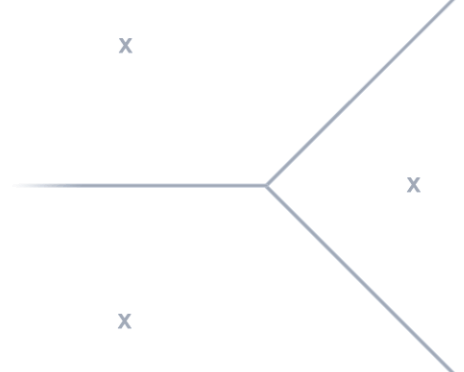
Abbildung 8.4: Die bereits bekannte Fermifunktion mit verschiedenen Variationen des Temperaturparameters.

Übungsaufgaben

Aufgabe 15. Geben Sie den Speicherbedarf für ein Hopfieldnetz mit $|K| = 1000$ Neuronen an, wenn die Gewichte $w_{i,j}$ als ganze Zahlen gespeichert werden sollen. Kann der Wertebereich der Gewichte eingegrenzt werden, um Speicher zu sparen?

Aufgabe 16. Berechnen Sie die Gewichte $w_{i,j}$ für ein Hopfieldnetz unter Verwendung der Trainingsmenge

$$\begin{aligned} P =&\{(-1,-1,-1,-1,-1,1);\\ &(-1,1,1,-1,-1,-1);\\ &(1,-1,-1,1,-1,1)\}. \end{aligned}$$



Kapitel 9

Learning Vector Quantization

Learning Vector Quantization ist ein Lernverfahren mit dem Ziel, in vordefinierte Klassen unterteilte Trainingsmengen von Vektoren durch wenige Repräsentanten-Vektoren möglichst gut wiederzugeben. Ist dies geschafft, so ist eine einfache Zuordnung bis dato unbekannter Vektoren in eine dieser Klassen möglich.

Allmählich neigt sich Teil II dieser Arbeit dem Ende entgegen – und so möchte ich auch ein Kapitel für den Abschluss dieses Teils schreiben, welches einen schönen Übergang darstellt: Ein Kapitel über die von Teuvo Kohonen beschriebene ***Learning Vector Quantization*** (Kurz: ***LVQ***) [Koh89], welche man als verwandt mit den *Self Organizing Feature Maps* bezeichnen kann. Diese werden direkt im Anschluss beschrieben, nämlich im nächsten Kapitel, welches sich bereits in Teil III der Arbeit befindet, da SOMs unüberwacht lernen. Ich möchte also nach der Untersuchung von LVQ Abschied vom überwachten Lernen nehmen.

Gleich im Voraus möchte ich ankündigen, dass es verschiedene Variationen von LVQ gibt, die ich zwar erwähnen, aber nicht genau darstellen werde – es ist mehr das Ziel dieses Kapitels, das zugrundeliegende Prinzip zu erforschen.

9.1 Über Quantisierung

Um die *Learning Vector Quantization* zu erforschen, sollten wir uns zunächst klarmachen, was denn Quantization, zu Deutsch ***Quantisierung*** ist, die man auch mit ***Diskretisierung*** bezeichnen kann.

Jeder von uns kennt den diskreten Zahlenraum

$$\mathbb{N} = \{1, 2, 3, \ldots\},$$

in dem die natürlichen Zahlen liegen. ***Diskret*** bedeutet, dass dieser Raum aus voneinander *abgetrennten* Elementen besteht, die nicht miteinander verbunden sind. In unserem Beispiel sind diese Elemente eben diese Zahlen, denn in den natürlichen Zahlen gibt es ja beispielsweise keine Zahl zwischen 1 und 2. ***Kontinuierlich*** ist dagegen beispielsweise der Raum der reellen Zahlen $\mathbb{R}$: Egal, wie nah man daraus zwei Zahlen wählt, es wird immer eine Zahl zwischen ihnen geben.

Quantisierung bedeutet nun die Unterteilung eines kontinuierlichen Raums in diskrete Abschnitte: Indem man der reellen Zahl 2.71828 beispielsweise alle Nachkommastellen entfernt, könnte man diese Zahl der natürlichen Zahl 2 zuweisen. Hierbei ist klar, dass jede andere Zahl mit einer 2 vor dem Komma ebenfalls der natürlichen Zahl 2 zugewiesen würde, die 2 wäre also eine Art *Repräsentant* für alle reellen Zahlen im Intervall $[2; 3)$.

Zu beachten ist, dass wir einen Raum auch unregelmäßig quantisieren können: So wäre der Zeitstrahl einer Woche beispielsweise in Arbeitstage und Wochenende quantisierbar.

Ein Spezialfall der Quantisierung ist die ***Digitalisierung***: Im Fall der Digitalisierung sprechen wir immer von einer *gleichmäßigen* Quantisierung eines kontinuierlichen Raums in ein Zahlensystem zu einer bestimmten ***Basis***. Geben wir beispielsweise Zahlen in den Computer ein, werden diese in das Binärsystem (Basis 2) digitalisiert.

Definition 9.1 (Quantisierung)**.** Unterteilung eines kontinuierlichen Raums in diskrete Abschnitte.

Definition 9.2 (Digitalisierung)**.** Gleichmäßige Quantisierung.

9.2 LVQ unterteilt den Eingaberaum in separate Bereiche

Nun können wir schon fast anhand des Namens beschreiben, was LVQ uns ermöglichen soll: Es soll einen Eingaberaum durch eine Menge von repräsentativen Vektoren in Klassen unterteilen, die ihn möglichst gut wiedergeben (Abb. 9.1 auf der rechten Seite). Jedes Element des Eingaberaums soll also einem Vektor als Repräsentanten, also einer Klasse zugeordnet werden können, wobei die Menge dieser Repräsentanten den ganzen Eingaberaum möglichst genau repräsentieren soll. Einen solchen Vektor nennt man ***Codebookvektor***. Ein Codebookvektor ist dann der Repräsentant genau derjenigen

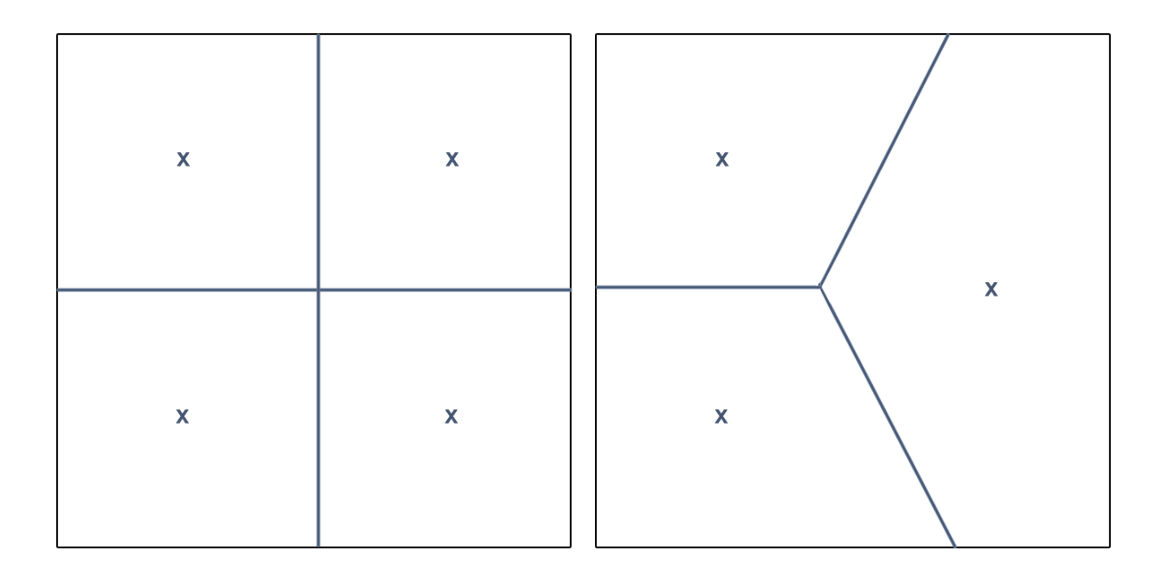


Abbildung 9.1: Beispielquantisierungen eines zweidimensionalen Eingaberaums. Die Linien stellen die Klassengrenzen dar, die $\times$ markieren die Codebookvektoren.

Vektoren des Eingaberaums, die am nächsten bei ihm liegen, was den Eingaberaum in besagte diskrete Bereiche unterteilt.

Hervorzuheben ist, dass wir im Voraus wissen müssen, wieviele Klassen wir haben und welches Trainingsbeispiel zu welcher Klasse gehört. Weiter ist wichtig, dass die Klassen nicht disjunkt sein müssen, sie dürfen sich also überlappen.

Solche Unterteilungen der Daten in Klassen sind interessant bei vielen Problemstellungen, bei denen es nützlich ist, anstatt der unter Umständen riesigen Ursprungsmenge von Daten nur ein paar typische Vertreter zu betrachten – sei es, weil es weniger Aufwand ist, oder weil es an Genauigkeit einfach genügt.

9.3 Benutzung von Codebookvektoren: Der nächste gewinnt

Die Benutzung eines einmal angefertigten Satzes Codebookvektoren ist sehr einfach: Für einen Eingabevektor y entscheidet man die Klassenzugehörigkeit ganz einfach daran, welcher Codebookvektor am nächsten zu ihm liegt – die Codebookvektoren bilden also ein ***Voronoidiagramm*** in unserer Eingabemenge. Da jeder Codebookvektor

eindeutig einer Klasse zuzuordnen ist, ist hierdurch jeder Eingabevektor einer Klasse zugeordnet.

9.4 Ausrichtung der Codebookvektoren

Wie wir schon angedeutet haben, handelt es sich bei LVQ um ein überwachtes Lernverfahren. Wir besitzen also einen Teaching Input, der dem Lernverfahren sagt, ob ein Eingangsmuster richtig oder falsch klassifiziert wurde: Anders ausgedrückt müssen wir bereits im Vorhinein die Anzahl der zu repräsentierenden Klassen bzw. die Anzahl der Codebookvektoren kennen.

Das Ziel des Lernvorganges ist also grob gesprochen, dass wir eine im Vorhinein bestimmte Anzahl zufällig initialisierter Codebookvektoren durch Trainingsbeispiele dazu bringen, die Trainingsdaten möglichst gut wiederzuspiegeln.

9.4.1 Vorgehensweise beim Lernen

Das Lernen funktioniert nach einem einfachen Schema. Man besitzt (da das Lernen überwacht ist) eine Menge P von $|P|$ vielen Trainingsbeispielen. Wie wir außerdem schon wissen, sind auch die Klassen vordefiniert, man besitzt also weiterhin eine Klassenmenge C. Jeder Klasse ist ein Codebookvektor eindeutig zugeordnet, wir können also sagen, dass die Klassenmenge $|C|$ viele Codebookvektoren $C_1, C_2, \ldots, C_{|C|}$ enthält.

Dies führt uns zum Aufbau der Trainingsbeispiele: Sie sind von der Form (p, c), enthalten also zum einen den Trainings-Eingabevektor p und zum anderen dessen Klassenzugehörigkeit c. Für die Klassenzugehörigkeit gilt hierbei

$$c \in \{1, 2, \ldots, |C|\},$$

sie ordnet also das Trainingsbeispiel eindeutig einer Klasse bzw. einem Codebookvektor zu.

Intuitiv könnte man zum Lernen nun sagen: „Wozu ein Lernverfahren? Wir rechnen den Durchschnitt aller Klassenmitglieder aus, platzieren dort deren Codebookvektor und gut.“ Dass unser Lernverfahren aber wesentlich mehr macht, werden wir gleich sehen.

Wir wollen nun kurz die Schritte des grundsätzlichen LVQ-Lernverfahrens betrachten:

Initialisierung: Wir platzieren unseren Satz Codebookvektoren auf zufällige Orte im Eingaberaum.

Trainingsbeispiel: Ein Trainingsbeispiel p aus unserer Trainingsmenge P wird gewählt und präsentiert.

Abstandsmessung: Wir messen den Abstand $||p - C||$ aller Codebookvektoren $C_1, C_2, \ldots, C_{|C|}$ zu unserer Eingabe p.

Gewinner: Der naheliegendste Codebookvektor gewinnt, also derjenige mit

$$\min_{C_i \in C} ||p - C_i||.$$

Lernvorgang: Der Lernvorgang findet durch die Regel

$$\Delta C_i = \eta(t) \cdot h(p, C_i) \cdot (p - C_i) \quad (9.1)$$

$$C_i(t+1) = C_i(t) + \Delta C_i \quad (9.2)$$

statt, die wir nun aufschlüsseln wollen.

- ▷ Der erste Faktor $\eta(t)$ ist, wie wir schon oft gesehen haben, eine zeitabhängige Lernrate, die es uns ermöglicht, zwischen großen Lernschritten und Fine-Tuning zu differenzieren.
- ▷ Der letzte Faktor $(p - C_i)$ ist offensichtlich die *Richtung*, in die wir den Codebookvektor verschieben.
- ▷ Kernstück aber ist die Funktion $h(p, C_i)$: Sie trifft eine Fallunterscheidung.

  Zuweisung richtig: Der Gewinnervektor ist der Codebookvektor der Klasse, der p zugehörig ist. In diesem Fall liefert die Funktion positive Werte, der Codebookvektor bewegt sich auf das p zu.

  Zuweisung falsch: Der Gewinnervektor repräsentiert nicht die Klasse, der p zugehörig ist. Er bewegt sich daher von p weg.

Wir sehen, dass wir die Funktion h nicht genau definiert haben. Aus gutem Grund: Ab hier teilt sich LVQ in verschiedene Nuancen auf, abhängig davon wie exakt h und die Lernrate bestimmt sein sollen (genannt ***LVQ1***, ***LVQ2***, ***LVQ3***, ***OLVQ***, etc). Die Unterschiede liegen beispielsweise in der Stärke der Codebookvektor-Bewegungen. Sie haben aber alle das gleiche hier dargestellte Grundprinzip gemeinsam, und wie angekündigt möchte ich sie nicht weiter betrachten – insofern schreibe ich auch keine formalen Definitionen zu obiger Lernregel und LVQ an sich auf.

9.5 Verbindung zu Neuronalen Netzen

Bis jetzt kann man sich trotz des Lernvorganges fragen, was denn LVQ mit Neuronalen Netzen zu tun hat. Man kann die Codebookvektoren als Neuronen mit festem Ort im Inputraum ansehen, ähnlich wie bei RBF-Netzen. Zudem ist es in der Natur oft so, dass ein Neuron pro Gruppe feuern darf (ein Gewinnerneuron, hier ein Codebookvektor), während alle anderen von ihm inhibiert werden.

Ich setze das kurze Kapitel über Learning Vector Quantization an diese Stelle im Script, weil wir im folgenden Kapitel über Self Organizing Maps den Ansatz weiterverfolgen wollen: Wir werden weitere Eingaben mithilfe von im Eingangsraum verteilten Neuronen klassifizieren, nur, dass wir diesmal nicht wissen, welche Eingabe zu welcher Klasse gehört.

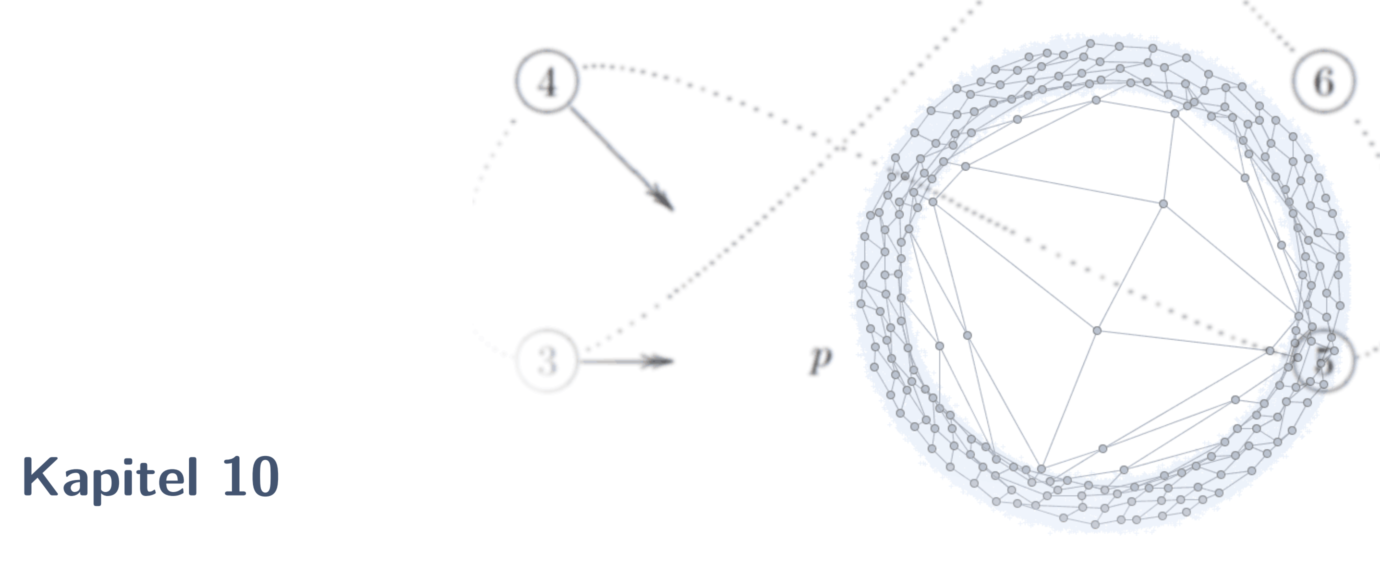
Kommen wir nun zu den *unüberwacht lernenden Netzen*!

Übungsaufgaben

Aufgabe 17. Geben Sie eine Quantisierung an, die im 5-dimensionalen Einheitswürfel $\mathcal{H}$ alle Vektoren $H \in \mathcal{H}$ gleichmäßig in eine von 1024 Klassen einteilt.

Teil III

Unüberwacht lernende Netzparadigmen



Kapitel 10

Self Organizing Feature Maps

Ein Paradigma unüberwacht lernender Neuronaler Netze, welches einen Eingaberaum durch seine feste Topologie kartographiert und so selbstständig nach Ähnlichkeiten sucht. Funktion, Lernverfahren, Variationen und Neuronales Gas.

Betrachtet man die Konzepte biologischer Neuronaler Netze in der Einleitung, so kann man sich die Frage stellen, wie denn unser Gehirn die Eindrücke, die es täglich erhält, speichert und abruft – hier sei hervorgehoben, dass das Gehirn keine Trainingsbeispiele, also keine „gewünschte Ausgabe“ hat, und wo wir schon darüber nachdenken, merken wir, dass auch keine Ausgabe in diesem Sinne vorhanden ist: Unser Gehirn ist nicht wie eine Abbildung oder ein Perceptron, das Eingabewerte auf Ausgabewerte abbildet. Unser Gehirn reagiert auf Eingaben von außen mit Zustandsänderungen – diese sind sozusagen seine Ausgabe.

Basierend auf diesem Grundsatz und der Frage nachgehend, wie biologische Neuronale Netze sich selbst organisieren können, schuf Teuvo Kohonen in den 1980er Jahren seine *Self Organizing Feature Maps* [Koh82,Koh98], kurz ***Self Organizing Maps*** oder ***SOMs*** genannt – ein Paradigma Neuronaler Netze, in dem der Zustand des Netzes die Ausgabe ist, und das vollkommen unüberwacht, also ohne Teacher lernt.

Im Unterschied zu anderen Netzparadigmen, welche wir bereits kennengelernt haben, stellt man bei SOMs keine Fragen danach, was die Neurone berechnen – man fragt nur, *welches Neuron gerade aktiv ist.* Dies ist biologisch sehr gut motiviert: Sind in der Biologie Neurone mit bestimmten Muskeln verbunden, interessiert es in aller Regel weniger, wie stark ein bestimmter Muskel kontrahiert wird, sondern *welcher* Muskel angesteuert wird. Anders ausgedrückt: Es interessiert nicht, was genau Neurone ausgeben, sondern *welches Neuron* etwas ausgibt. SOMs sind also wesentlich biologieverwandter als z.B. die FeedForward-Netze, die wir vermehrt für Rechenaufgaben nutzen.

10.1 Aufbau einer Self Organizing Map

SOMs haben – wie das Gehirn – typischerweise die Aufgabe, einen hochdimensionalen Input (N Dimensionen) auf Bereiche in einem niedrigdimensionalen ***Gitter*** (G Dimensionen) abzubilden, also sozusagen eine Karte von dem hochdimensionalen Raum zu zeichnen. Um diese Karte zu erschaffen, erhält die SOM einfach beliebig viele Punkte aus dem Inputraum. Die SOM wird während der Eingabe der Punkte versuchen, die Orte, auf denen die Punkte auftreten, so gut wie möglich mit ihren Neuronen abzudecken. Dies bedeutet insbesondere, dass jedes Neuron einem bestimmten Ort im Inputraum zuzuordnen ist.

Diese Fakten sind zunächst etwas verwirrend, man muss sie kurz durchdenken. Es gibt also zwei Räume, in denen SOMs arbeiten:

- Den N-dimensionalen Eingaberaum und
- das G-dimensionale Gitter, auf dem die Neurone liegen bzw. das die Nachbarschaftsbeziehungen der Neurone und damit die *Netztopologie* angibt.

Bei einem eindimensionalen Gitter könnten die Neurone beispielweise wie an einer Perlenkette aufgereiht sein, jedes Neuron würde genau zwei Nachbarn besitzen (bis auf die beiden End-Neurone). Ein zweidimensionales Gitter könnte eine rechtwinklige Anordnung von Neuronen sein (Abb. 10.1 auf der rechten Seite). Eine weitere denkbare Anordnung im zweidimensionalen wäre in einer Art Wabenform. Auch ungleichmäßige Topologien sind möglich, wenn auch nicht sehr häufig. Topologien mit mehr Dimensionen und wesentlich mehr Nachbarschaftsbeziehungen wären auch denkbar, werden aber aufgrund der mangelnden Visualisierungsfähigkeit nicht oft eingesetzt.

Auch dann, wenn $N = G$ gilt, sind die beiden Räume nicht gleich und müssen unterschieden werden – sie haben in diesem Spezialfall nur die gleiche Dimension.

Wir werden die Funktionsweise einer Self Organizing Map nun zunächst kurz formal betrachten und dann an einigen Beispielen klarmachen.

Definition 10.1 (SOM-Neuron)**.** Ähnlich den Neuronen in einem RBF-Netz besitzt ein ***SOM-Neuron*** k eine feste Position c_k (ein ***Zentrum***) im Eingaberaum.

Definition 10.2 (Self Organizing Map)**.** Eine Self Organizing Map ist eine Menge K von SOM-Neuronen. Bei Eingabe eines Eingabevektors wird genau dasjenige Neuron $k \in K$ aktiv, welches dem Eingabemuster im Eingaberaum am nächsten liegt. Die Dimension des Eingaberaumes nennen wir N.

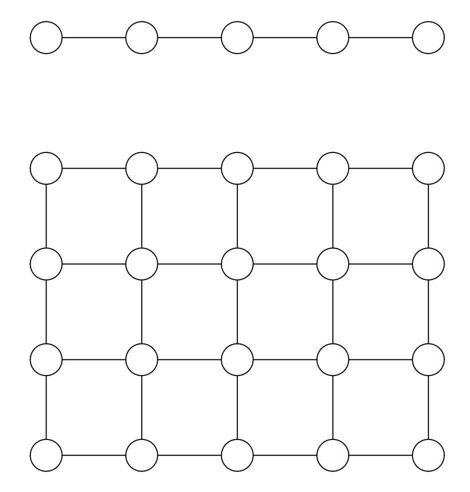


Abbildung 10.1: Beispieltopologien einer Self Organizing Map – Oben eine eindimensionale Topologie, unten eine zweidimensionale.

Definition 10.3 (Topologie)**.** Die Neurone sind untereinander durch Nachbarschaftsbeziehungen verbunden. Diese Nachbarschaftsbeziehungen nennen wir ***Topologie***. Die Topologie nimmt starken Einfluss auf das Training einer SOM. Sie wird durch die *Topologiefunktion* $h(i, k, t)$ definiert, wobei i das Gewinnerneuron[1] ist, k das gerade zu adaptierende Neuron (hierzu kommen wir später noch) und t der Zeitschritt. Wir bezeichnen die Dimension der Topologie mit G.

10.2 SOMs aktivieren immer das Neuron, was der Eingabe am nächsten liegt

Wie viele andere Neuronale Netze, muss die SOM erst trainiert werden, bevor man sie benutzt. Betrachten wir aber vor dem Training noch die sehr einfache Funktionsweise einer fertigen Self Organizing Map, da sie einige Analogien zum Training besitzt. Die Funktionsweise setzt sich aus folgenden Schritten zusammen:

Eingabe eines beliebigen Wertes p aus dem Eingangsraum $\mathbb{R}^N$.

1 Wir werden noch erfahren, was ein Gewinnerneuron ist.

Abstandsberechnung von jedem Neuron k zu p durch eine Norm – also Berechnung von $||p - c_k||$.

Ein Neuron wird aktiv, nämlich genau das Neuron i mit dem kürzesten oben berechneten Abstand zur Eingabe – alle anderen Neurone sind nicht aktiv. Dieses Paradigma der Aktivität heißt auch *Winner-Takes-All-Schema*. Die Ausgabe, welche wir zu einer Eingabe von einer SOM erwarten, ist, *welches* Neuron aktiv wird.

Viele Literaturstellen beschreiben die SOM wesentlich formaler: Es wird oft eine Eingabeschicht beschrieben, die in Richtung einer SOM-Schicht vollverknüpft ist. Die Eingabeschicht (N Neurone) leitet dann alle Eingaben an die SOM-Schicht weiter. Die SOM-Schicht ist in sich lateral vollverknüpft, so dass sich ein Gewinnerneuron herausbilden und die anderen Neurone inhibieren kann. Ich finde diese Art, eine SOM zu beschreiben, nicht sehr anschaulich und habe versucht, den Netzaufbau hier anschaulicher zu beschreiben.

Nun ist die Frage, welches Neuron bei welcher Eingabe aktiv wird – und genau dies ist die Frage, die uns das Netz während des Trainings von alleine beantwortet.

10.3 Training bringt die SOM-Topologie dazu, den Eingaberaum abzudecken

Das Training einer SOM ist ähnlich überschaubar wie die eben beschriebene Funktionsweise. Es gliedert sich im Wesentlichen in fünf Schritte, die teilweise deckungsgleich mit denen der Funktionsweise sind.

Initialisierung: Start des Netzes mit zufälligen Neuronenzentren $c_k \in \mathbb{R}^N$ aus dem Eingangsraum.

Anlegen eines Eingangsmusters: Es wird ein ***Stimulus***, also ein Punkt p aus dem Eingangsraum $\mathbb{R}^N$ gewählt. Dieser Stimulus wird nun in das Netz eingegeben.

Abstandsmessung: Für jedes Neuron k im Netz wird nun der Abstand $||p - c_k||$ bestimmt.

Winner takes all: Es wird das ***Gewinnerneuron*** i ermittelt, welches den kleinsten Abstand zu p besitzt, das also der Bedingung

$$||p - c_i|| \leq ||p - c_k|| \quad \forall\, k \neq i$$

genügt. Wie aus der Bedingung ersichtlich, kann man bei mehreren Gewinnerneuronen eines nach Belieben wählen.

Adaption der Zentren: Die Zentren der Neurone werden innerhalb des Eingangsraumes nach der Vorschrift[2]

$$\Delta c_k = \eta(t) \cdot h(i, k, t) \cdot (p - c_k),$$

versetzt, wobei die Werte Δc_k einfach auf die bisherigen Zentren addiert werden. Aus dem letzten Faktor wird bereits offensichtlich, dass die Ortsänderung der Neurone k proportional zu der Entfernung zum eingegebenen Muster p und wie gewohnt zu einer zeitabhängigen Lernrate $\eta(t)$ ist. Die oben besprochene *Topologie* des Netzes nimmt ihren Einfluss durch die Funktion $h(i, k, t)$, die wir im Folgenden erforschen werden.

Definition 10.4 (SOM-Lernregel)**.** Eine SOM wird trainiert, indem ihr ein Eingabemuster präsentiert und das *Gewinnerneuron* dazu ermittelt wird. Das Gewinnerneuron und seine durch die Topologiefunktion definierten Nachbarneuronen adaptieren dann ihre Zentren nach der Vorschrift

$$\Delta c_k = \eta(t) \cdot h(i, k, t) \cdot (p - c_k), \tag{10.1}$$

$$c_k(t + 1) = c_k(t) + \Delta c_k(t). \tag{10.2}$$

10.3.1 Die Topologiefunktion bestimmt, wie stark ein lernendes Neuron seine Nachbarn beeinflusst

Die ***Topologiefunktion*** h ist nicht auf dem Eingangsraum, sondern *auf dem Gitter* definiert und stellt die Nachbarschaftsbeziehungen zwischen den Neuronen dar – also die Topologie des Netzes. Sie kann zeitabhängig sein (und ist es auch oft) – dies erklärt den Parameter t. Der Parameter k ist der durch alle Neurone laufende Index, und der Parameter i ist der Index des Gewinnerneurons.

Prinzipiell ist der Sinn der Funktion, einen großen Wert anzunehmen, falls k Nachbar des Gewinners oder gar der Gewinner selbst ist, und kleine Werte, falls nicht. Schärfer definiert: Die Topologiefunktion muss *unimodal* sein, also genau ein Maximum besitzen – dieses Maximum muss beim Gewinnerneuron i liegen, das zu sich selbst natürlich die Entfernung 0 hat.

Zusätzlich macht es uns die Zeitabhängigkeit beispielsweise möglich, die Nachbarschaft mit der Zeit schrumpfen zu lassen.

---

2 Achtung: Viele Quellen schreiben diese Vorschrift $\eta h(p - c_k)$, was dem Leser fälschlicherweise suggeriert, dass es sich bei h um eine Konstante handelt. Dieses Problem ist einfach lösbar, indem man die Multiplikationspunkte $\cdot$ nicht weglässt.

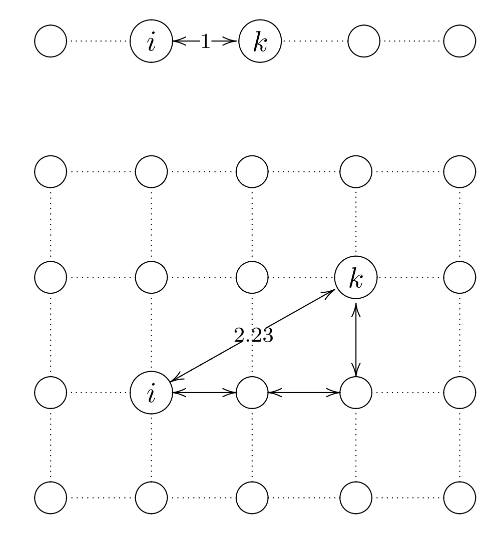


Abbildung 10.2: Beispiel-Abstände einer eindimensionalen SOM-Topologie (oben) und einer zweidimensionalen SOM-Topologie (unten) zwischen zwei Neuronen i und k. Im unteren Fall bilden wir den Euklidischen Abstand (im zweidimensionalen äquivalent zum Satz des Pythagoras). Im oberen Fall wird einfach die diskrete Weglänge zwischen i und k abgezählt. In beiden Fällen habe ich der Einfachheit halber eine feste Gitterkantenlänge von 1 gefordert.

Um große Werte für Nachbarn von i und kleine Werte für Nicht-Nachbarn ausgeben zu können, braucht die Funktion h eine Art *Abstandsbegriff* auf dem Gitter, irgendwoher muss sie also wissen, *wie weit* i und k auf dem Gitter voneinander entfernt sind. Es gibt verschiedene Methoden zur Berechnung des Abstandes.

Hierfür könnte bei einem zweidimensionalen Gitter z.B. der *euklidische Abstand* (unterer Teil der Abb. 10.2) verwendet werden oder bei einem eindimensionalen Gitter einfach die Anzahl der Verbindungen zwischen den Neuronen i und k (oberer Teil derselben Abbildung).

Definition 10.5 (Topologiefunktion)**.** Die Topologiefunkton $h(i, k, t)$ beschreibt die Nachbarschaftsbeziehungen in der Topologie. Sie kann eine beliebige unimodale Funktion sein, die maximal wird, wenn $i = k$ gilt. Eine Zeitabhängigkeit ist optional, wird aber oft verwendet.

10.3.1.1 Vorstellung gängiger Abstands- und Topologiefunktionen

Eine gängige Abstandsfunktion wäre beispielsweise die uns schon bekannte ***Gaußglocke*** (siehe auch Abb. 10.3 auf der folgenden Seite). Sie ist unimodal mit einem Maximum bei 0, zusätzlich kann sie durch ihren Parameter σ in ihrer Breite verändert werden, was wir für die Realisierung der mit der Zeit schrumpfenden Nachbarschaft nutzen können: Wir beziehen die Zeitabhängigkeit einfach auf das σ, erhalten also ein monoton sinkendes $\sigma(t)$. Unsere Topologiefunktion könnte dann wie folgt aussehen:

$$h(i,k,t) = \mathrm{e}^{\left(-\frac{||g_i - g_k||^2}{2\cdot\sigma(t)^2}\right)}, \tag{10.3}$$

wobei hier g_i und g_k die Positionen der Neurone *auf dem Gitter* sind, nicht im Eingaberaum, welche wir mit c_i und c_k bezeichnen würden.

Weitere Funktionen, die anstatt der Gaußfunktion eingesetzt werden können, sind zum Beispiel die ***Kegelfunktion***, die ***Zylinderfunktion*** oder die ***Mexican-Hat-Funktion*** (Abb. 10.3 auf der folgenden Seite). Die Mexican-Hat-Funktion bietet hierbei eine besondere biologische Motivation: Sie stößt durch ihre negativen Stellen manche Neurone in der Umgebung des Gewinnerneurons ab, was man auch in der Natur schon beobachtet hat. Dies kann für schärfere Abgrenzung der Kartenbereiche sorgen – genau aus diesem Grund wurde sie auch von Teuvo Kohonen selbst vorgeschlagen. Diese Regulierungseigenschaft ist aber für die Funktion der Karte an sich nicht notwendig, es kann sogar passieren, dass dadurch die Karte divergiert, also gewissermaßen explodiert.

10.3.2 Lernraten und Nachbarschaften können über die Zeit monoton sinken

Damit in den späteren Phasen des Trainings nicht mehr die ganze Karte sehr stark in Richtung eines neuen Musters gezogen wird, wird bei den SOMs oft mit zeitlich monoton sinkenden Lernraten und Nachbarschaftsgrößen gearbeitet. Sprechen wir zunächst über die Lernrate: Typische Größenordnungen für den Zielwert der Lernrate sind zwei Größenordnungen kleiner als der Startwert, beispielsweise könnte gelten

$$0.01 < \eta < 0.6.$$

Diese Größe ist aber auch wieder abhängig zu machen von der Netztopologie bzw. der Größe der Nachbarschaft.

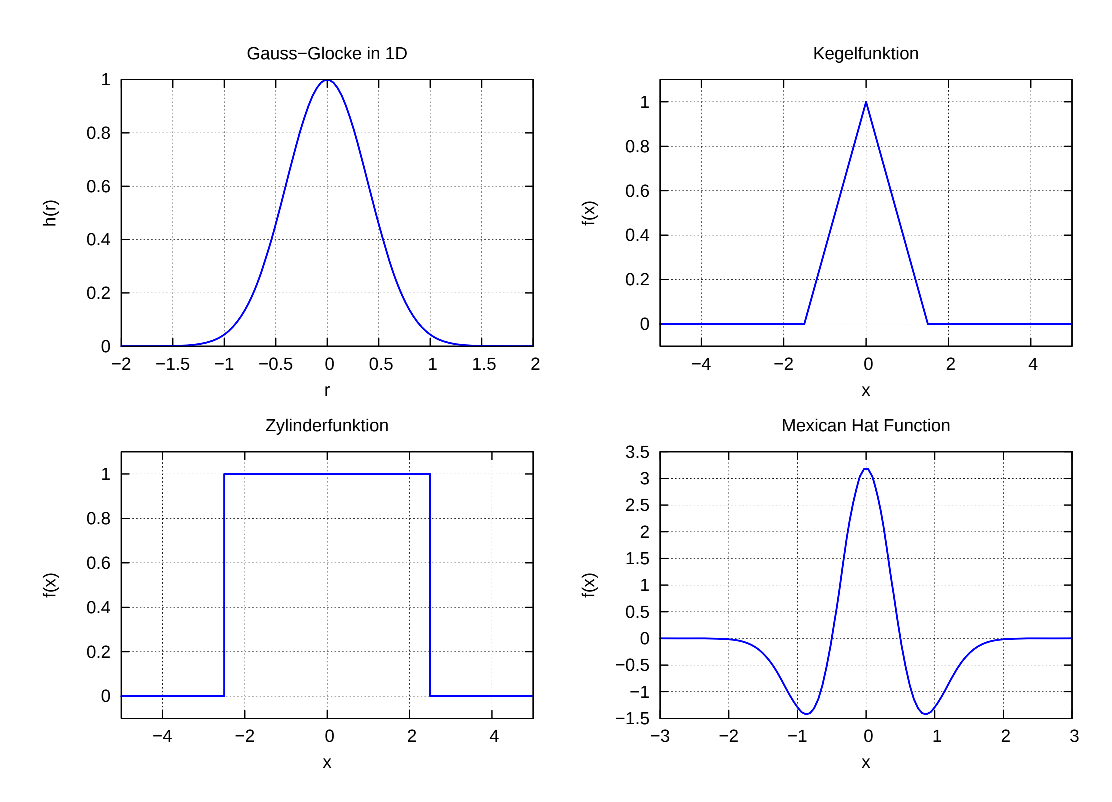


Abbildung 10.3: Gaußglocke, Kegelfunktion, Zylinderfunktion und die von Kohonen vorgeschlagene Mexican-Hat-Funktion als Beispiele für Topologiefunktionen einer SOM.

Eine sinkende Nachbarschaftsgröße kann, wie wir gesehen haben, beispielsweise mit einem zeitabhängig monoton sinkenden σ bei Benutzung der Gaußglocke in der Topologiefunktion realisiert werden.

Der Vorteil bei einer sinkenden Nachbarschaftsgröße ist, dass ein sich bewegendes Neuron zu Anfang viele Neurone in seiner Umgebung „mitzieht“, sich das zufällig initialisierte Netz also am Anfang schnell und sauber entfalten kann. Zum Ende des Lernvorganges hin werden nur noch wenige Neurone auf einmal beeinflusst, was das Netz im Gesamten steifer macht, aber ein gutes „fine tuning“ der einzelnen Neurone ermöglicht.

Zu beachten ist weiterhin, dass stets

$$h \cdot \eta \leq 1$$

gelten muss, sonst würden Neurone andauernd am aktuellen Trainingsbeispiel vorbeischießen.

Doch nun genug der Theorie – betrachten wir eine SOM im Einsatz!

10.4 Beispiele für die Funktionsweise von SOMs

Beginnen wir mit einem einfachen, im Kopf nachvollziehbaren Beispiel.

In diesem Beispiel verwenden wir einen zweidimensionalen Eingangsraum, es gilt also $N = 2$. Die Gitterstruktur sei eindimensional ($G = 1$). Weiterhin soll unsere Beispiel-SOM aus insgesamt 7 Neuronen bestehen und die Lernrate sei $\eta = 0.5$.

Auch unsere Nachbarschaftsfunktion halten wir recht einfach, um das Netz im Kopf nachvollziehen zu können:

$$h(i, k, t) = \begin{cases} 1 & k \text{ direkter Nachbar von } i, \\ 1 & k = i, \\ 0 & \text{sonst.} \end{cases} \tag{10.4}$$

Wir betrachten jetzt das soeben beschriebene Netz mit zufälliger Initialisierung der Zentren (Abb. 10.4 auf der folgenden Seite) und geben ein Trainingsmuster p ein. Offensichtlich liegt das Eingangsmuster in unserem Beispiel dem Neuron 3 am nächsten – dieses ist also der Gewinner.

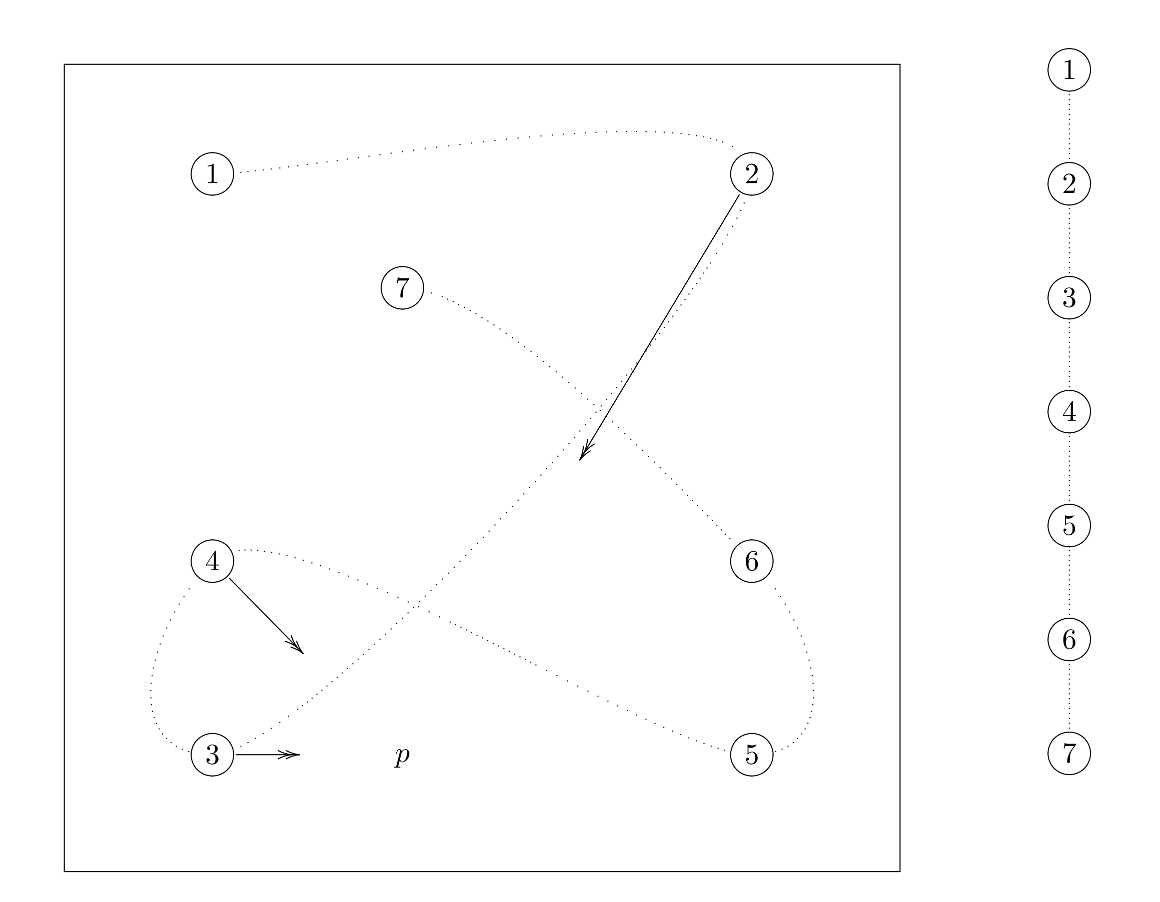


Abbildung 10.4: Darstellung des zweidimensionalen Eingaberaumes (links) und des eindimensionalen Topologieraumes (rechts) einer Self Organizing Map. Neuron 3 ist das Gewinnerneuron, da es p am nächsten liegt. Die Nachbarn von 3 in der Topologie sind die Neurone 2 und 4. Die Pfeile markieren die Bewegung des Gewinnerneurons und seiner Nachbarn in Richtung des Trainingsbeispiels p.

Die eindimensionale Topologie des Netzes ist hier zur Veranschaulichung durch die gepunkteten Linien in den Eingangsraum aufgetragen. Die Pfeile markieren die Bewegung des Gewinners und seiner Nachbarn auf das Muster zu.

Wir erinnern uns an die Lernvorschrift für SOMs

$$\Delta c_k = \eta(t) \cdot h(i, k, t) \cdot (p - c_k)$$

und arbeiten die drei Faktoren von hinten nach vorne ab:

Lernrichtung: Wir erinnern uns, dass die Neuronenzentren c_k Vektoren im Eingangsraum sind, genau wie das Muster p. Der Faktor $(p - c_k)$ gibt uns also den Vektor vom Neuron k zum Muster p an. Dieser wird nun mit verschiedenen Skalaren multipliziert:

Unsere Topologiefunktion h besagt, dass nur das Gewinnerneuron und seine beiden nächsten Nachbarn (hier: 2 und 4) lernen dürfen, indem sie bei allen anderen Neuronen 0 ausgibt. Eine Zeitabhängigkeit ist nicht gegeben. Unser Vektor $(p - c_k)$ wird also entweder mit 1 oder mit 0 multipliziert.

Die Lernrate gibt, wie immer, die Stärke des Lernens an. Es gilt wie schon gesagt $\eta = 0.5$ – wir kommen also insgesamt auf das Ergebnis, dass sich das Gewinnerneuron und seine Nachbarn (hier: Die Neurone 2, 3 und 4) dem Muster p um die Hälfte des Weges nähern (dies markieren die Pfeile in der Abbildung).

Obwohl das Zentrum von Neuron 7 vom Eingangsraum aus gesehen wesentlich näher am Eingangsmuster p liegt als das Neuron 2, lernt das Neuron 2, und das Neuron 7 nicht. Ich möchte daran noch einmal deutlich machen, dass die Netztopologie bestimmt, welches Neuron mitlernen darf, *und nicht die Lage im Eingangsraum.* Dies ist genau der Mechanismus, durch den eine Topologie einen Eingangsraum aussagekräftig abdecken kann, ohne mit ihm auf irgendeine Weise verwandt sein zu müssen.

Nach der Adaption der Neurone 2, 3 und 4 wird das nächste Muster angelegt, und so weiter, und so fort. Ein weiteres Beispiel, wie sich eine solche eindimensionale SOM im zweidimensionalen Inputraum bei gleichverteilten Inputmustern über die Zeit entwickeln kann, sehen wir an der Abbildung 10.5 auf Seite 193.

Endzustände von ein- und zweidimensionalen SOMs bei verschieden geformten Inputräumen sehen wir in der Abbildung 10.6 auf Seite 194. Wie wir sehen, sind nicht alle Inputräume durch jede Netztopologie schön abdeckbar, es gibt sogenannte *freiliegende* Neurone – Neurone, welche in einem Bereich liegen, in dem kein Inputmuster je aufgetreten ist. Eine eindimensionale Topologie produziert in der Regel weniger freiliegende Neurone als eine zweidimensionale: Beispielsweise beim Training auf ringförmig angeordnete Eingabemuster ist es bei einer zweidimensionalen quadratischen Topologie so gut wie unmöglich, die freiliegenden Neurone in der Mitte des Rings zu verhindern – diese werden ja während des Trainings in jede Richtung gezogen, so dass sie schlussendlich einfach in der Mitte bleiben. Das macht die eindimensionale Topologie aber keineswegs

zur Optimaltopologie, da sie nur weniger komplexe Nachbarschaftsbeziehungen finden kann als eine mehrdimensionale.

10.4.1 Topologische Defekte sind Fehlentfaltungen der SOM

Es kann während des Entfaltens einer SOM vorkommen, dass diese einen ***Topologischen Defekt*** (Abb. 10.7 auf Seite 195) bildet, sich also nicht richtig entfaltet. Ein Topologischer Defekt kann am besten mit dem Wort „Verknotung“ beschrieben werden.

Ein Mittel gegen Topologische Defekte kann sein, die Startwerte für die Nachbarschaftsgröße zu erhöhen, denn je komplexer die Topologie ist (bzw. je mehr Nachbarn jedes Neuron besitzt, man könnte ja auch dreidimensionale oder wabenförmige zweidimensionale Topologien erzeugen) desto schwerer ist es für eine zufällig initialisierte Karte, sich zu entfalten.

10.5 Man kann die Auflösung einer SOM ortsabhängig dosieren

Wir haben gesehen, dass eine SOM trainiert wird, indem hintereinander immer wieder Eingabemuster aus dem Eingangsraum $\mathbb{R}^N$ eingegeben werden, so dass sich die SOM an diesen Mustern ausrichten und diese *kartographieren* wird. Es kann nun vorkommen, dass wir eine bestimmte Untermenge U des Eingaberaums genauer kartographiert haben möchten als den Rest.

Die Lösung dieses Problems kann mit SOMs denkbar einfach realisiert werden: Wir präsentieren der SOM während des Trainings ganz einfach überproportional viele Eingabemuster aus diesem Bereich U. Werden der SOM mehr Trainingsmuster aus $U \subset \mathbb{R}^N$ präsentiert als aus dem Rest $\mathbb{R}^N \setminus U$, so werden sich dort auch mehr Neurone zusammenfinden, während sich die restlichen Neurone auf $\mathbb{R}^N \setminus U$ dünner verteilen (Abb. 10.8 auf Seite 196).

Wie in der Abbildung ersichtlich, kann so der Rand der SOM etwas deformiert werden – dies ist ausgleichbar, indem man dem Rand des Inputraumes ebenfalls eine etwas höhere Wahrscheinlichkeit zuweist, von Trainingsmustern getroffen zu werden (eine Taktik, die man sowieso oft anwendet, um mit den SOMs besser „in die Ecken zu kommen“).

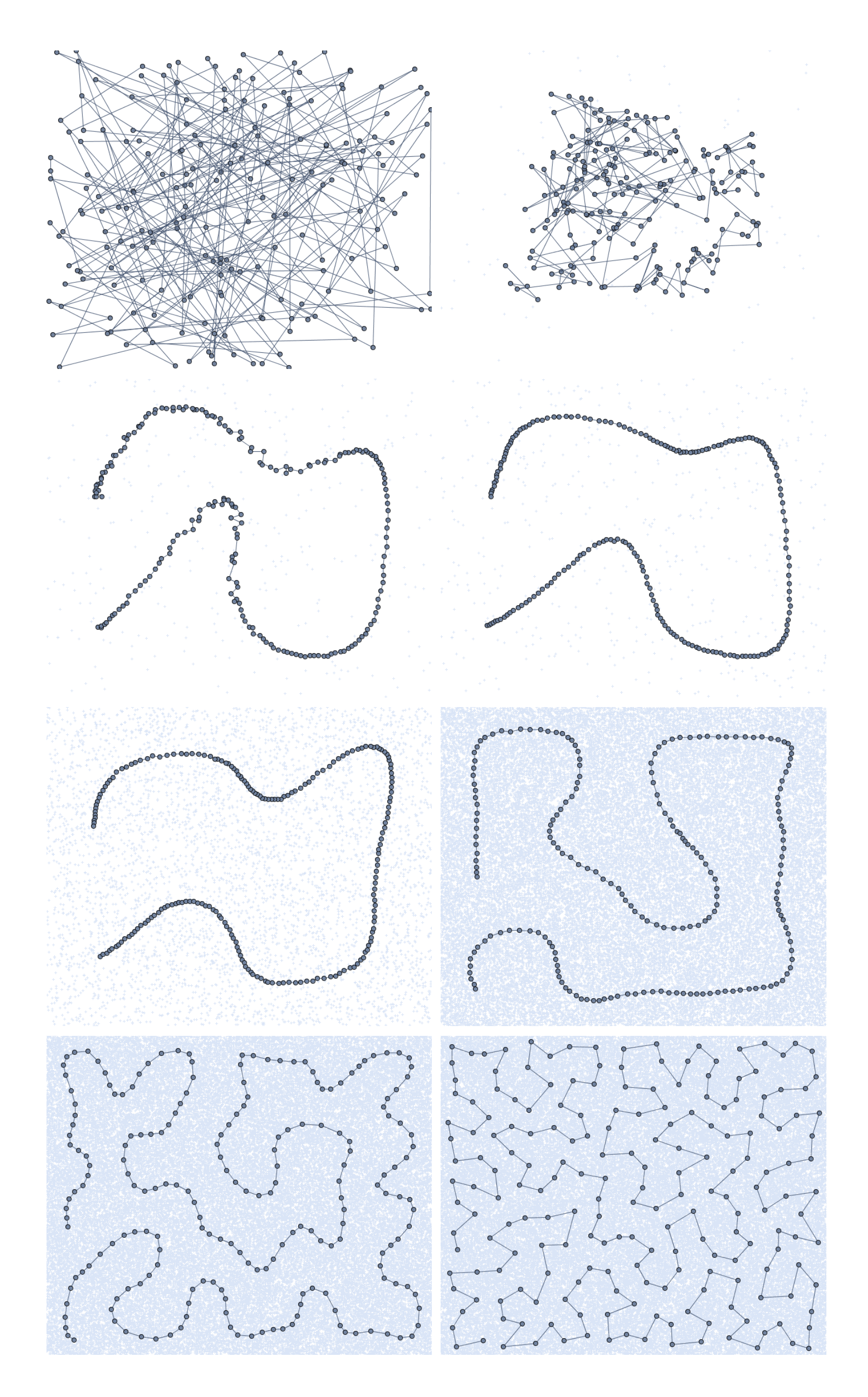


Abbildung 10.5: Verhalten einer SOM mit eindimensionaler Topologie ($G = 1$) nach Eingabe von 0, 100, 300, 500, 5000, 50000, 70000 und 80000 zufällig verteilten Eingabemustern $p \in \mathbb{R}^2$. η fiel während des Trainings von 1.0 auf 0.1, der σ-Parameter der als Nachbarschaftsmaß eingesetzten Gauß-Funktion von 10.0 auf 0.2.

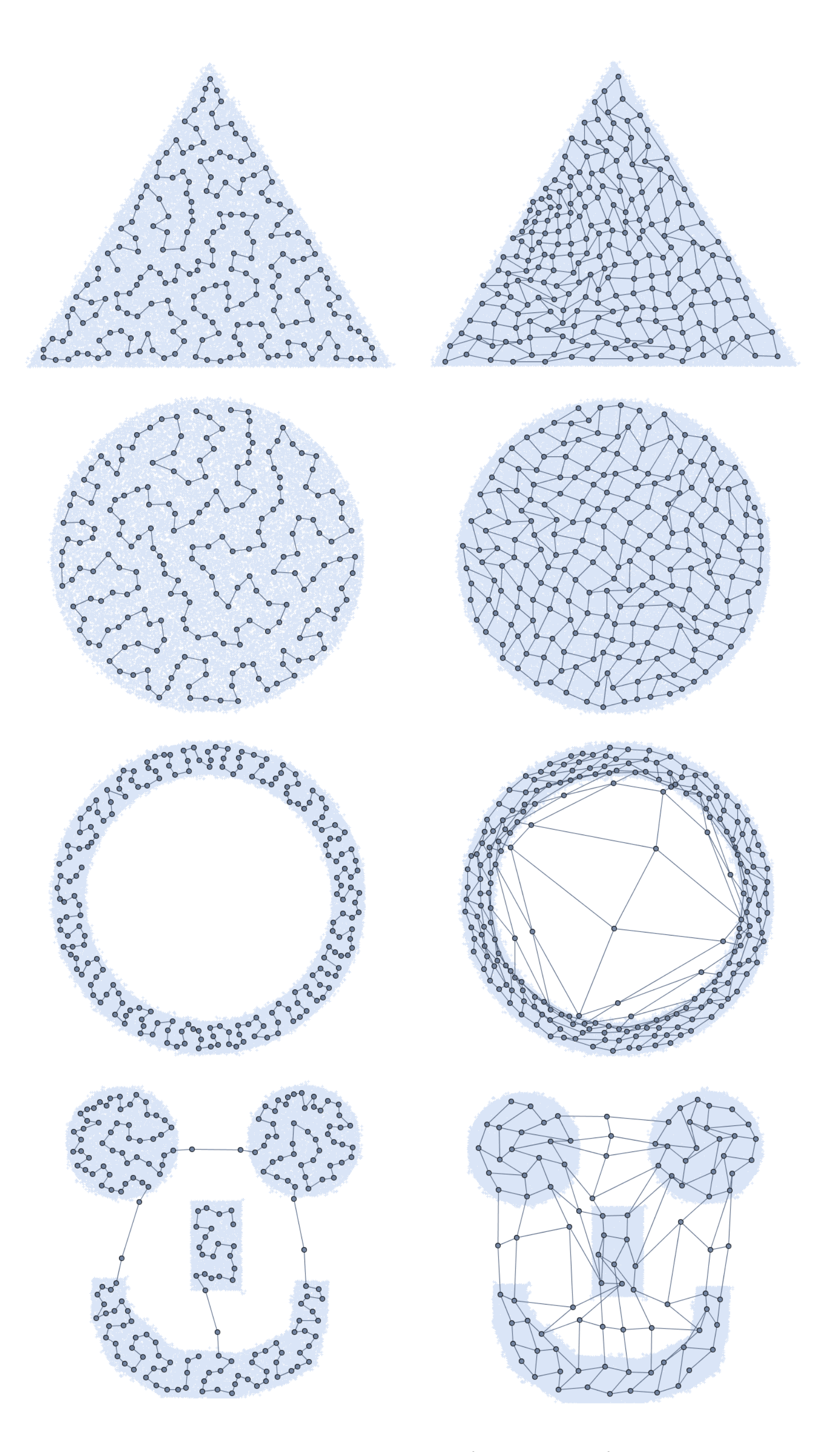


Abbildung 10.6: Endzustände von eindimensionalen (linke Spalte) und zweidimensionalen (rechte Spalte) SOMs auf verschieden abgedeckten Inputräumens. Genutzt wurden bei eindimensionaler Topologie 200 Neurone, bei zweidimensionaler 10×10 Neurone und bei allen Karten 80.000 Eingabemuster.

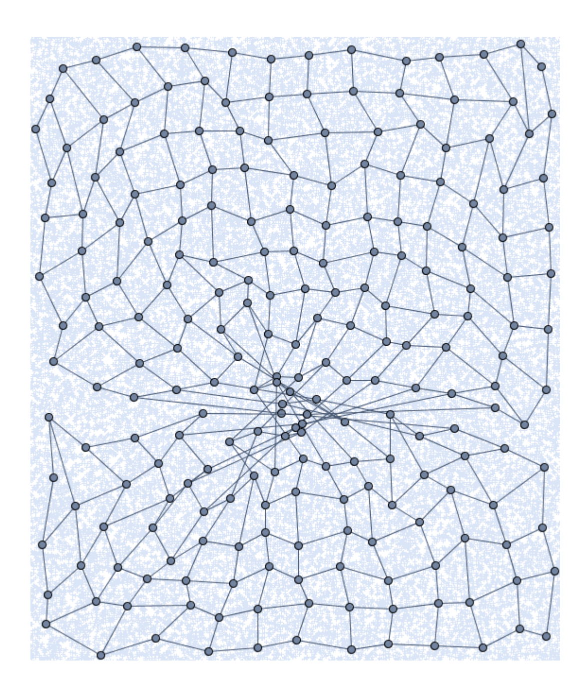


Abbildung 10.7: Ein Topologischer Defekt in einer zweidimensionalen SOM.

Auch wird oft bei den Rand- bzw. Eckneuronen eine höhere Lernrate angewandt, da sie durch die Topologie nur nach innen gezogen werden - auch dies sorgt für eine signifikant bessere Eckabdeckung.

10.6 Anwendung von SOMs

Self Organizing Maps und ihre Variationen haben eine Vielzahl von Anwendungsgebieten in Bezug auf die biologisch inspirierte ***assoziative Speicherung*** von Daten.

Beispielsweise ist es gelungen, auf einer SOM mit einer zweidimensionalen diskreten Gittertopologie die verschiedenen Phoneme der Finnischen Sprache abzubilden und so Nachbarschaften zu finden (eine SOM macht ja nichts anderes, als Nachbarschaftsbeziehungen zu finden). Man versucht also wieder, einen hochdimensionalen Raum auf einen niederdimensionalen Raum (die Topologie) herabzubrechen, schaut, ob sich irgendwelche Strukturen herausbilden – und siehe da: Es bilden sich klar definierte Bereiche für die einzelnen Phoneme heraus.

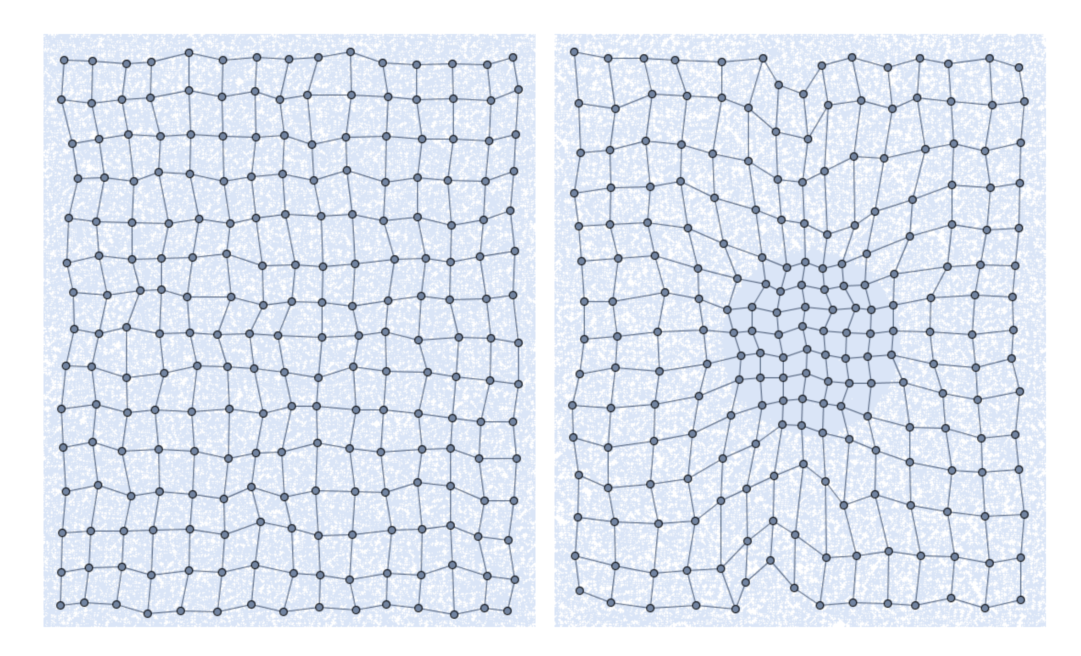


Abbildung 10.8: Training einer SOM mit $G = 2$ auf einen zweidimensionalen Inputraum. Links war die Wahrscheinlichkeit für jede Koordinate des Inputraumes, zum Trainingsmuster zu werden, gleich. Rechts war sie für den zentralen Kreis im Inputraum mehr als zehnmal so groß wie für den Rest des Inputraumes (sichtbar an der größeren Musterdichte im Hintergrund). Offensichtlich drängen sich die Neurone in diesem Kreis viel stärker und der restliche Bereich ist etwas undichter abgedeckt, aber immer noch jeweils einigermaßen gleichmäßig. Beide SOMs wurden mit 80.000 Trainingsbeispielen und abfallendem η $(1 \to 0.2)$ sowie abfallendem σ $(5 \to 0.5)$ trainiert.

Teuvo Kohonen selbst hat sich die Mühe gemacht, eine Vielzahl von Papern, welche seine SOMs erwähnen, nach Schlüsselwörtern zu durchsuchen. In diesem großen Eingangsraum besetzen jetzt die einzelnen Paper einzelne Positionen, je nach Schlüsselwortvorkommen. Kohonen hat dann eine SOM mit $G = 2$ kreiert und damit den von ihm erstellten hochdimensionalen „Paper-Raum“ durch sie kartographieren lassen.

Man kann also ein Paper, an dem man Gefallen findet, in die fertig trainierte SOM eingeben und schauen, welches Neuron in der SOM davon aktiviert wird. Wahrscheinlich wird man feststellen, dass man an den in der Topologie *benachbarten* Papern auch Gefallen findet. Diese Art von Gehirn-ähnlicher ***kontextbasierter Suche*** funktioniert noch bei vielen anderen Inputräumen.

Festzuhalten ist, dass das System selbst festlegt, was innerhalb der Topologie benachbart, also *ähnlich* ist – genau das macht es so interessant.

An diesem Beispiel ist direkt ersichtlich, dass die Lage c der Neurone im Inputraum nicht aussagekräftig ist. Interessant ist es vielmehr, nun zu schauen, welches Neuron bei Eingabe eines bis jetzt unbekannten Inputmusters aktiv wird. Man kann als nächstes schauen, bei welchen bisherigen Eingaben genau dieses Neuron ebenfalls aktiv war – und hat sofort eine Gruppe von einander sehr ähnlichen Eingaben entdeckt. Je weiter nun Eingaben innerhalb der Topologie auseinander liegen, um so weniger Gemeinsamkeiten haben sie. Die Topologie bildet also quasi eine Karte der Merkmale der Eingaben – reduziert auf anschaulich wenige Dimensionen im Vergleich zur Inputdimension.

Oftmals ist die Topologie einer SOM daher zweidimensional, da sie sich so sehr gut visualisieren lässt, während der Eingaberaum sehr hochdimensional sein kann.

10.6.1 Mit SOMSs kann man Zentren für RBF-Neurone finden

SOMs richten sich genau auf die Orte der ausgehenden Eingaben aus – demzufolge werden sie z.B. gerne genutzt, um die Zentren eines RBF-Netzes zu wählen. Das Paradigma der RBF-Netze haben wir bereits in Kapitel 6 kennengelernt.

Wie wir bereits gesehen haben, kann man sogar steuern, welche Bereiche des Inputraums mit höherer Auflösung abgedeckt werden sollen – oder im Zusammenhang mit RBF-Netzen gesprochen, welche Bereiche unserer Funktion das RBF-Netz mit mehr Neuronen, also genauer bearbeiten soll. Als zusätzliches nützliches Merkmal der Kombination aus RBF-Netz und SOM kann man die durch die SOM erhaltene Topologie nutzen, um beim schlussendlichen Training eines Neurons des RBF-Netzes „benachbarte“ RBF-Neurone auf verschiedene Weisen mit zu beeinflussen.

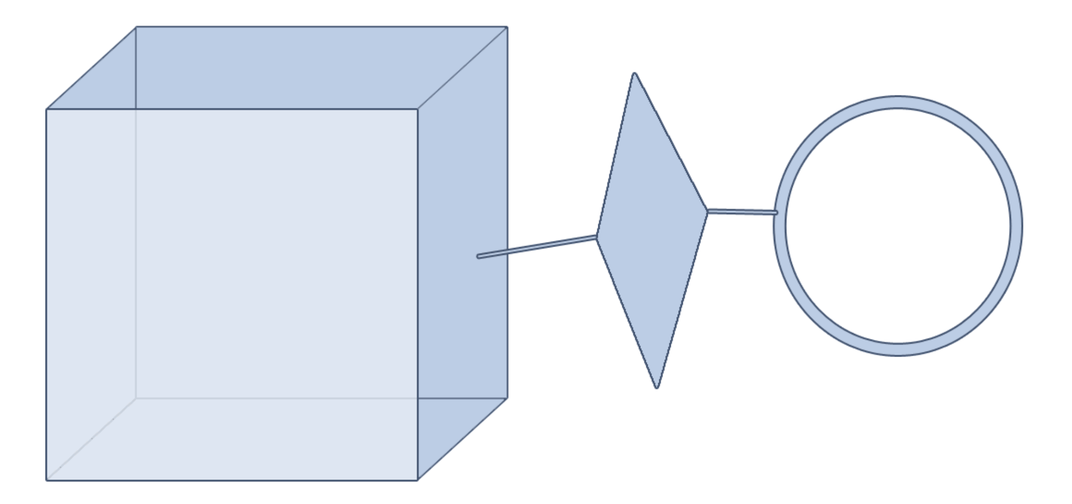


Abbildung 10.9: Eine Figur, die verschiedene Unterräume des eigentlichen Eingaberaums zu verschiedenen Orten ausfüllt und daher durch eine SOM nur schwer ausgefüllt werden kann.

Viele Simulatoren Neuronaler Netze bieten hierfür eine zusätzliche sogenannte *SOM-Schicht* im Zusammenhang mit der Simulation von RBF-Netzen an.

10.7 Variationen von SOMs

Für verschiedene Variationen der Repräsentationsaufgaben gibt es auch verschiedene Variationen der SOMs:

10.7.1 Ein Neuronales Gas ist eine SOM ohne feste Topologie

Das ***Neuronale Gas*** ist eine Variation der Self Organizing Maps von Thomas Martinetz [MBS93], die aus der Schwierigkeit heraus entstanden ist, komplizierte Inputformationen abzubilden, welche sich teilweise nur in Unterräumen des Inputraums abspielen oder die Unterräume sogar wechseln (Abb. 10.9).

Die Idee von einem Neuronalen Gas besteht grob gesprochen darin, ein SOM ohne Gitterstruktur zu realisieren. Die Lernschritte sehen aufgrund der Abstammung von den SOMs den SOM-Lernschritten auch sehr ähnlich, haben aber einen zusätzlichen Zwischenschritt:

- Wieder zufällige Initialisierung der $c_k \in \mathbb{R}^n$
- Wahl und Präsentation eines Musters aus dem Eingangsraum $p \in \mathbb{R}^n$
- Neuronenabstandsmessung
- Ermittlung des Gewinnerneurons i
- *Zwischenschritt:* Bilden einer nach der Entfernung zum Gewinnerneuron aufsteigend sortierten Liste L von Neuronen. Erstes Neuron in der Liste L ist also das Neuron, was dem Gewinnerneuron am *nächsten* liegt.
- Ändern der Zentren durch die bekannte Vorschrift, aber der leicht modifizierten Topologiefunktion
$$h_L(i, k, t).$$

Die gegenüber der ursprünglichen Funktion $h(i, k, t)$ modifizierte Funktion $h_L(i, k, t)$ sieht nun jeweils die ersten Elemente der Liste als Nachbarschaft des Gewinnerneurons i an. Dies hat direkt zur Folge, dass sich – ähnlich der frei schwebenden Moleküle in einem Gas – die Nachbarschaftsbeziehungen zwischen den Neuronen jederzeit ändern können, auch die Anzahl der Nachbarn ist nahezu beliebig. Die Entfernung innerhalb der Nachbarschaft wird nun durch die Entfernung innerhalb des Eingaberaums repräsentiert.

Die Masse der Neurone kann durch eine stetig sinkende Nachbarschaftsgröße genauso versteift werden, wie eine SOM – sie hat aber keine feste Dimension, sondern kann jeweils die Dimension annehmen, die lokal gerade gebraucht wird, was sehr vorteilhaft sein kann.

Von Nachteil kann sein, dass nicht durch ein festes Gitter erzwungen wird, dass der Inputraum relativ gleichmäßig abgedeckt wird, und so Löcher in der Abdeckung entstehen oder Neurone vereinsamen können.

Es bleibt wie immer in der Pflicht des Anwenders, diese Arbeit bei allen Praxistipps nicht als Katalog für Patentlösungen zu verstehen, sondern selbst die Vor- und Nachteile zu erforschen.

Anders als bei einer SOM muss sich die Nachbarschaft in einem Neuronalen Gas anfangs auf die Gesamtheit aller Neurone beziehen, weil sonst einige Ausreißer der zufälligen Initialisierung vielleicht nie in die Nähe der restlichen Gruppe kommen. Dies zu vergessen, ist ein sehr beliebter Fehler bei der Implementierung eines Neuronalen Gases.

Mit einem Neuronalen Gas kann man auch eine sehr komplexe Inputform lernen, da wir nicht an ein festdimensionales Gitter gebunden sind. Rechenaufwändig werden

kann aber das dauernde Sortieren der Liste (hier kann es viel bringen, die Liste gleich in einer von sich aus geordneten Datenstruktur abzulegen). Ein Beispiel für solche Inputformen findet sich in Abb. 10.9 auf Seite 198.

Definition 10.6 (Neuronales Gas)**.** Ein Neuronales Gas unterscheidet sich von einer SOM durch eine vollständig dynamische Nachbarschaftsfunktion. In jedem Lernzyklus wird neu entschieden, welche Neurone die Nachbarneurone des Gewinnerneurons sind. In der Regel ist das Kriterium für diese Entscheidung die Entfernung der Neurone zum Gewinnerneuron im Inputraum.

10.7.2 Eine Multi-SOM besteht aus mehreren separaten SOMs

Um eine weitere Variante der SOMs zu präsentieren, möchte ich eine erweiterte Problemstellung formulieren: Was machen wir bei Eingabemustern, von denen wir vorher wissen, dass sie sich in verschiedene (vielleicht disjunkte) Bereiche abgrenzen?

Die Idee hierzu ist, nicht nur eine SOM zu nehmen, sondern gleich mehrere: Eine ***Multi Self Organizing Map***, kurz ***M-SOM*** [GKE01b, GKE01a, GS06]. Die SOMs müssen nicht die gleiche Topologie oder Größe haben, eine M-SOM ist nur ein Zusammenschluss aus M vielen SOMs.

Der Lernvorgang funktioniert analog zu den einfachen SOMs – allerdings werden nur die Neurone, die zum Gewinner-SOM eines jeden Trainingsschritts gehören, adaptiert. Mit zwei SOMs ist es also einfach, zwei disjunkte Cluster Daten zu repräsentieren, selbst wenn einer davon nicht in allen Dimensionen des Eingangsraumes $\mathbb{R}^N$ vertreten ist. Die einzelnen SOMs spiegeln sogar genau die Cluster wieder.

Definition 10.7 (Multi-SOM)**.** Eine Multi-SOM ist nichts weiter als eine gleichzeitige Verwendung von M vielen SOMs.

10.7.3 Ein Multi-Neuronales Gas besteht aus mehreren separaten neuronalen Gasen

Analog zum Multi-SOM haben wir hier wieder eine Menge von M vielen Neuronalen Gasen: ein ***Multi-Neuronales Gas*** [GS06, SG06]. Dieses Konstrukt verhält sich analog zu Neuronalem Gas und M-SOM: Es werden wieder nur die Neurone adaptiert, die im Gewinner-Gas sind.

Der Leser wird sich natürlich fragen, was denn ein Multi-Neuronales Gas für Vorteile bringt, da bereits ein einzelnes Neuronales Gas in der Lage ist, sich in Cluster zu

zerteilen und auch auf komplizierten Inputmustern mit wechselnder Dimension zu arbeiten. Das ist zwar grundsätzlich richtig, doch es gibt zwei gravierende Vorteile eines Multi-Neuronalen Gases gegenüber einem einfachen Neuronalen Gas.

1. Bei mehreren Gasen kann man jedem Neuron direkt ansehen, zu welchem Gas es gehört. Das ist insbesondere bei Clustering-Aufgaben wichtig, für die Multi-Neuronale Gase in jüngster Zeit genutzt werden. Einfache Neuronale Gase können zwar auch Cluster finden und abdecken, man sieht aber nicht, welches Neuron jetzt zu welchem Cluster gehört.

2. Es spart enorm Rechenzeit, große Ursprungs-Gase in mehrere kleinere aufzuteilen, da (wie schon angesprochen) das Sortieren der Liste L sehr viel Rechenzeit in Anspruch nehmen kann, das Sortieren von mehreren kleineren Listen $L_1, L_2, \ldots, L_M$ aber weniger aufwändig ist – selbst dann, wenn diese Listen insgesamt genausoviele Neurone enthalten.

Man erhält zwar nur lokale Sortierungen und keine globale, diese reichen aber in den meisten Fällen aus.

Wir können nun zwischen zwei Extremfällen des Multi-Neuronalen Gases wählen: Der eine Extremfall ist das normale Neuronale Gas $M = 1$, wir verwenden also nur ein einziges Neuronales Gas. Interessanterweise verhält sich der andere Extremfall (sehr großes M, wenige oder nur ein Neuron pro Gas) analog zum K-Means-Clustering (siehe zum Thema Clusteringverfahren auch den Exkurs A).

Definition 10.8 (Multi-Neuronales Gas)**.** Ein Multi-Neuronales Gas ist nichts weiter als eine gleichzeitige Verwendung von M vielen Neuronalen Gasen.

10.7.4 Wachsende neuronale Gase können sich selbst Neurone hinzufügen

Ein ***Wachsendes Neuronales Gas*** ist eine Variation des beschriebenen Neuronalen Gases, dem nach bestimmten Regeln mehr und mehr Neurone hinzugefügt werden. Man versucht so, der Vereinsamung von Neuronen oder der Bildung großer Abdeckungslöcher entgegenzuwirken.

Es sei hier nur benannt, aber nicht weiter erforscht.

Eine wachsende SOM zu konstruieren ist insofern schwieriger, als dass neue Neurone in die Nachbarschaft eingegliedert werden müssen.

Übungsaufgaben

Aufgabe 18. Mit einem regelmäßigen zweidimensionalen Gitter soll eine zweidimensionale Fläche möglichst „gut“ ausgelegt werden.

1. Welche Gitterstruktur wäre hierfür am besten geeignet?
2. Welche Kriterien für „gut“ bzw. „am besten“ haben Sie verwendet?

Diese Aufgabe ist bewusst sehr schwammig formuliert.



Kapitel 11

Adaptive Resonance Theory

Ein ART-Netz in seiner ursprünglichen Form soll binäre Eingabevektoren klassifizieren, also einer 1-aus-n-Ausgabe zuordnen. Gleichzeitig sollen bis jetzt unklassifizierbare Muster erkannt und einer neuen Klasse zugeordnet werden.

Wie in anderen kleinen Kapiteln wollen wir wieder versuchen, die grundlegende Idee der ***Adaptive Resonance Theory*** (Kurz: ***ART***) zu ergründen, ohne ihre Theorie wirklich in der Tiefe zu betrachten.

Wir haben bereits in verschiedenen Abschnitten angedeutet, dass es schwierig ist, mit Neuronalen Netzen neue Information zusätzlich zu bestehender „hinzuzulernen“, ohne jedoch die bestehende Information zu zerstören – diesen Umstand bezeichnen wir als ***Stabilitäts-Plastizitäts-Dilemma***.

1987 veröffentlichten Stephen Grossberg und Gail Carpenter mit dem Ziel der Entschärfung dieser Problematik die erste Version ihres ART-Netzes [Gro76], der ab da eine ganze Familie von ART-Verbesserungen folgte (die wir ebenfalls noch kurz ansprechen wollen).

Es handelt sich um eine Idee des unüberwachten Lernens, die (zunächst binäre) Mustererkennung zum Ziel hatte, genauer gesagt, die Einteilung von Mustern in Klassen – wobei ein ART-Netz aber zusätzlich in der Lage sein soll, neue Klassen zu finden.

11.1 Aufgabe und Struktur eines ART-Netzes

Ein ART-Netz ist aufgebaut aus genau zwei Schichten: Der Eingabeschicht I und der Erkennungsschicht O, wobei die Eingabeschicht in Richtung der Erkennungsschicht vollverknüpft ist. Diese Vollverknüpfung gibt uns eine ***Top-Down-Gewichtsmatrix***

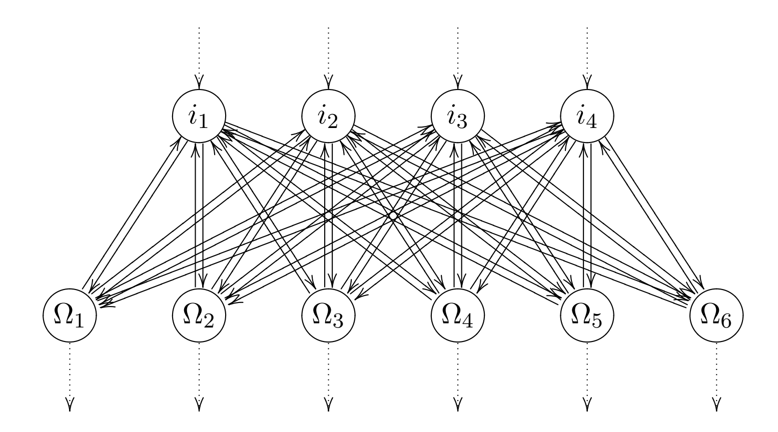


Abbildung 11.1: Vereinfachte Darstellung des Aufbaus eines ART-Netzes. Oben die Eingabeschicht, unten die Erkennungsschicht. Für die Abbildung außer Acht gelassen worden sind die laterale Inhibition der Erkennungsschicht und die Steuerungsneurone.

W an, die die Gewichtswerte von jedem Neuron der Eingabeschicht zu jedem Neuron der Erkennungsschicht enthält (Abb. 11.1).

An der Eingabeschicht werden einfach binäre Muster eingegeben, welche an die Erkennungsschicht weitergegeben werden – während die Erkennungsschicht eine 1-aus-$|O|$-Kodierung ausgeben soll, also dem Winner-Takes-All-Schema folgen soll. Um diese 1-aus-$|O|$-Kodierung zu realisieren, kann man beispielsweise das Prinzip der *lateralen Inhibition* nutzen – oder in der Implementierung pragmatischerweise per IF-Abfrage das am stärksten aktivierte Neuron suchen.

11.1.1 Resonanz erfolgt, indem Aktivitäten hin- und hergeworfen werden

Zusätzlich gibt es aber noch eine ***Bottom-Up-Gewichtsmatrix*** V, die die Aktivitäten in der Erkennungsschicht wieder in die Eingabeschicht propagiert. Es ist nun offensichtlich, dass diese Aktivitäten immer hin- und hergeworfen werden, was den Begriff der ***Resonanz*** ins Spiel bringt. Jede Aktivität der Eingabeschicht bewirkt eine Aktivität der Erkennungsschicht, während dort jede Aktivität wiederum eine Aktivität in der Eingabeschicht bewirkt.

Zusätzlich zu den zwei genannten Schichten existieren in einem ART-Netz noch einige wenige Neurone, welche Kontrollfunktionen ausüben wie z.B. eine Signalverstärkung. Deren Theorie soll hier nicht weiter betrachtet werden, da nur das grundlegende Prinzip der ART-Netze klar werden soll. Ich erwähne sie nur, um darzustellen, dass das Art-Netz nach einer Eingabe trotz der vorhandenen Rückkopplungen in einen stabilen Zustand gelangen wird.

11.2 Der Lernvorgang eines Artnetzes ist geteilt in Top-Down- und Bottom-Up-Lernen

Die Kunst der Adaptive Resonance Theory liegt nicht nur in der Gestaltung des ART-Netzes, sondern auch in ihrem Lernvorgang, der zweigeteilt ist: Wir trainieren zum einen die Top-Down-Matrix W und zum anderen die Bottom-Up-Matrix V (Abb. 11.2 auf Seite 207).

11.2.1 Mustereingabe und Top-Down-Lernen

Wenn ein Muster in das Netz eingegeben wird, sorgt es wie schon gesagt für eine Aktivierung an den Ausgabeneuronen, wobei das stärkste Neuron gewinnt. Anschließend werden die zum Ausgabeneuron gehenden Gewichte der Matrix W dahingehend verändert, dass die Ausgabe des stärksten Neurons Ω noch verstärkt wird. Die Klassenzugehörigkeit des Eingabevektors zu der Klasse des Ausgabeneurons Ω wird also verstärkt.

11.2.2 Resonanz und Bottom-Up-Lernen

Etwas tricky ist nun das Trainieren der rückwärtsgerichteten Gewichte der Matrix V: Es werden nur die Gewichte vom jeweilige Gewinnerneuron zur Eingabeschicht trainiert und als Teaching Input unser aktuell angelegtes Eingabemuster verwendet. Das Netz wird also darauf trainiert, eingegebene Vektoren zu verstärken.

11.2.3 Hinzufügen eines Ausgabeneurons

Es kann natürlich vorkommen, dass die Neurone ungefähr gleich aktiviert sind oder mehrere Neurone aktiviert sind, das Netz also unentschlossen ist. In diesem Fall wird durch die Mechanismen der Steuerungsneurone ein Signal ausgelöst, das ein neues

Ausgabeneuron hinzufügt. Das aktuelle Muster wird dann diesem Ausgabeneuron zugeordnet und die Gewichtssätze des neuen Neurons wie gewohnt trainiert.

Die Stärke des Systems liegt also nicht nur darin, Eingaben in Klassen zu unterteilen und neue Klassen zu finden – es kann uns bei Aktivierung eines Ausgabeneurons auch sagen, wie ein typischer Vertreter einer Klasse aussieht – ein bedeutendes Feature.

Oft kann das System Muster aber nur mäßig gut unterscheiden. Die Frage ist, wann einem neuen Neuron erlaubt wird, aktiv zu werden und wann gelernt werden soll. Auch um diese Fragen zu beantworten, existieren in einem ART-Netz verschiedene zusätzliche Kontrollneurone, die diese Fragen nach verschiedenen mathematischen Regeln beantworten und dafür zuständig sind, Spezialfälle abzufangen.

Hier benennen wir einen der größten Kritikpunkte an ART: Ein ART-Netz verwendet eine Spezialfallunterscheidung ähnlich einer IF-Abfrage, die man in den Mechanismus eines Neuronalen Netzes gepresst hat.

11.3 Erweiterungen

Wie schon eingangs erwähnt, wurden die ART-Netze vielfach erweitert.

ART-2 [CG87] ist eine Erweiterung auf kontinuierliche Eingaben und bietet zusätzlich (in einer ***ART-2A*** genannten Erweiterung) Verbesserungen der Lerngeschwindigkeit, was zusätzliche Kontrollneurone und Schichten zur Folge hat.

ART-3 [CG90] verbessert die Lernfähigkeit von ART-2, indem zusätzliche biologische Vorgänge wie z.B. die chemischen Vorgänge innerhalb der Synapsen adaptiert werden[1].

Zusätzlich zu den beschriebenen Erweiterungen existieren noch viele mehr.

1 Durch die häufigen Erweiterungen der Adaptive Resonance Theory sprechen böse Zungen bereits von „ART-n-Netzen“.

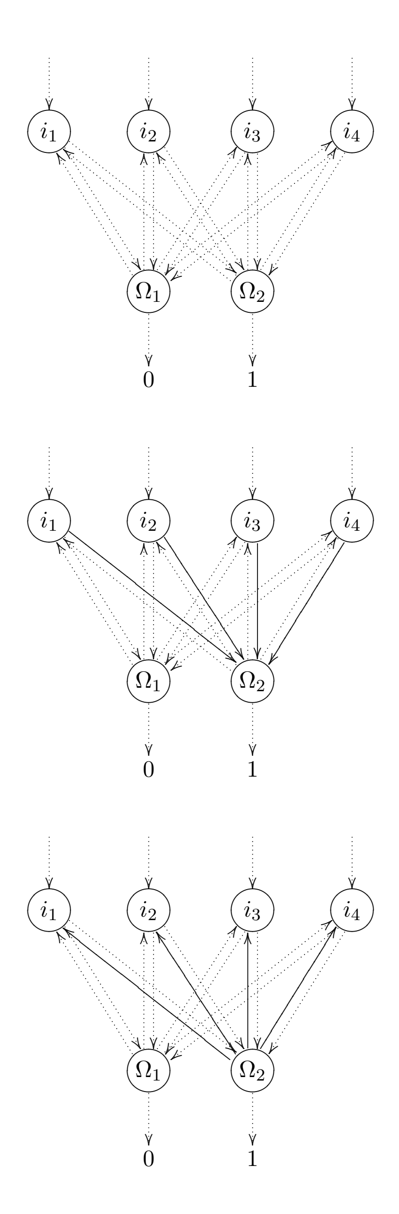


Abbildung 11.2: Vereinfachte Darstellung des zweigeteilten Trainings eines ART-Netzes: Die jeweils trainierten Gewichte sind durchgezogen dargestellt. Nehmen wir an, ein Muster wurde in das Netz eingegeben und die Zahlen markieren Ausgaben. **Oben:** Wie wir sehen, ist Ω_2 das Gewinnerneuron. **Mitte:** Also werden die Gewichte zum Gewinnerneuron hin trainiert und **(unten)** die Gewichte vom Gewinnerneuron zur Eingangsschicht trainiert.

Teil IV

Exkurse, Anhänge und Register

Anhang A

Exkurs: Clusteranalyse und Regional and Online Learnable Fields

Das Grimmsche Wörterbuch beschreibt das heute ausgestorbene deutsche Wort „Kluster“ mit „was dicht und dick zusammensitzet“. In der statistischen Clusteranalyse wird die Gruppenbildung innerhalb von Punktwolken untersucht. Vorstellung einiger Verfahren, Vergleich ihrer Vor- und Nachteile. Betrachtung eines lernenden Clusteringverfahrens auf Basis Neuronaler Netze. Ein Regional and Online Learnable Field modelliert aus einer Punktwolke mit womöglich sehr vielen Punkten eine vergleichsweise kleine Menge von für die Punktwolke repräsentativen Neuronen.

Wie schon angedeutet, lassen sich viele Problemstellungen auf Probleme der ***Clusteranalyse*** zurückführen. Dies macht die Erforschung von Verfahren notwendig, die untersuchen, ob sich innerhalb von Punktwolken Gruppen (sog. ***Cluster***) herausbilden. Da Verfahren zur Clusteranalyse für ihre Arbeit einen Abstandsbegriff zwischen zwei Punkten brauchen, muss auf dem Raum, in dem sich die Punkte finden, eine ***Metrik*** definiert sein. Wir wollen also kurz darauf eingehen, was eine Metrik ist.

Definition A.1 (Metrik)**.** Eine Relation $\mathrm{dist}(x_1, x_2)$, die für zwei Objekte x_1, x_2 definiert ist, heißt Metrik, wenn jedes der folgenden Kriterien gilt:

1. $\mathrm{dist}(x_1, x_2) = 0$ genau dann, wenn $x_1 = x_2$,
2. $\mathrm{dist}(x_1, x_2) = \mathrm{dist}(x_2, x_1)$, also Symmetrie,
3. $\mathrm{dist}(x_1, x_3) \leq \mathrm{dist}(x_1, x_2) + \mathrm{dist}(x_2, x_3)$, d. h. die Dreiecksungleichung gilt.

Umgangssprachlich ausgedrückt, ist eine Metrik ein Werkzeug, Abstände von Punkten in einem beliebig gearteten Raum festzustellen. Abstände müssen hierbei symmetrisch

sein, und der Abstand zwischen zwei Punkten darf nur 0 sein, wenn beide Punkte gleich sind. Zudem muss die Dreiecksungleichung gelten.

Metriken liefert uns beispielsweise der ***quadratische Abstand*** und der ***Euklidische Abstand***, welche wir schon kennengelernt haben. Aufbauend auf eine solche Metrik kann man Clusteringverfahren definieren, die eine Metrik als Abstandsmaß verwenden.

Wir wollen nun verschiedene Clusteringverfahren vorstellen und kurz diskutieren.

A.1 k-Means Clustering teilt Daten in eine vordefinierte Anzahl Cluster ein

k-Means Clustering nach J. MacQueen [Mac67] ist ein Algorithmus, der aufgrund seiner niedrigen Rechen- und Speicherkomplexität häufig verwendet wird und allgemein als „billig und gut“ angesehen wird. Der Algorithmus k-Means Clustering hat folgenden Arbeitsablauf:

1. Daten bereitstellen, die untersucht werden sollen
2. k, die Anzahl der Clusterzentren, definieren
3. k viele zufällige Vektoren für die Clusterzentren wählen (auch ***Codebookvektoren*** genannt).
4. Jeden Datenpunkt dem nächsten Codebookvektor[1] zuordnen
5. Clusterschwerpunkte für alle Cluster berechnen
6. Codebookvektoren auf neue Clusterschwerpunkte setzen.
7. Bei 4 weitermachen, bis keine Zuordnungsänderungen mehr eintreten.

Bei Punkt 2 sieht man schon eine der großen Fragen des k-Means-Algorithmus: Man muss im Vorhinein selbst die Anzahl k der Clusterzentren bestimmen, dies nimmt einem das Verfahren also nicht ab. Das Problem ist, dass man im Vorhinein nicht unbedingt weiß, wie k am besten bestimmt werden kann. Ein weiteres Problem ist, dass das Verfahren recht instabil werden kann, wenn die Codebookvektoren schlecht initialisiert werden. Da dies aber zufällig geschieht, hilft oft schon ein Neustart des Verfahrens, das zudem nicht sehr rechenaufwändig ist, was wiederum ein Vorteil ist. Verwenden Sie es im Bewusstsein dieser Schwächen, und Sie werden recht gute Ergebnisse erhalten.

1 Der Name *Codebookvektor* entstand, weil die oft verwendete Bezeichnung *Clustervektor* zu missverständlich war.

Komplizierte Strukturen, wie „Cluster in Clustern“ können allerdings nicht erkannt werden. Bei einem hoch gewählten k würde der äußere Ring dieser Konstruktion in der Abbildung als viele einzelne Cluster erkannt, bei einer niedrigen Wahl von k würde der Ring mit dem kleinen innenliegenden Cluster als ein Cluster gesehen.

Siehe für eine Veranschaulichung den oberen rechten Teil der Abb. A.1 auf Seite 215.

A.2 k-Nearest Neighbouring sucht die k nächsten Nachbarn jeden Datenpunkts

Das ***k-Nearest Neighbouring-Verfahren*** [CH67] verbindet jeden Datenpunkt mit den jeweils k vielen nächsten Nachbarn, was oft eine Unterteilung in Gruppen zur Folge hat. Eine solche Gruppe bildet dann einen Cluster. Der Vorteil ist hier, dass die Clusteranzahl von alleine entsteht – der Nachteil ein recht großer Speicher- und Rechenaufwand, um die nächten Nachbarn zu finden (es muss der Abstand von jedem zu jedem Datenpunkt ausgerechet und gespeichert werden).

Es gibt außerdem Spezialfälle, in denen das Verfahren bei zu großer Wahl von k Datenpunkte zusammenschließt, die eigentlich in verschiedene Cluster gehören (siehe die beiden kleinen Cluster oben rechts in der Abbildung). Cluster, die nur aus einem einzigen Datenpunkt bestehen, werden grundsätzlich mit einem anderen Cluster zusammengeschlossen, auch das ist nicht immer gewollt.

Weiterhin müssen die Bindungen unter den Punkten nicht symmetrisch sein.

Das Verfahren ermöglicht es aber, Ringe und somit auch „Cluster in Clustern“ zu erkennen, ein eindeutiger Vorteil. Weiterer Vorteil ist, dass das Verfahren adaptiv auf die Entfernungen in und zwischen Clustern eingeht.

Siehe für eine Veranschaulichung den unteren linken Teil der Abb. A.1.

A.3 ε-Nearest Neighbouring sucht für jeden Datenpunkt Nachbarn im Radius ε

Ein anderer Ansatz des Neighbourings: Hier geht die Nachbarschaftsfindung nicht über eine feste Anzahl k von Nachbarn, sondern über einen Radius ε – daher der Name ***Epsilon-Nearest Neighbouring***. Punkte, die maximal ε weit voneinander entfernt sind, sind Nachbarn. Hier ist der Speicher- und Rechenaufwand augenscheinlich wieder sehr hoch, was ein Nachteil ist.

Achtung, auch hier gibt es Spezialfälle: Zwei an sich getrennte Cluster können hier durch die ungünstige Lage eines einzigen Datenpunkts einfach verbunden werden. Dies kann zwar auch beim k-Nearest Neighbouring passieren, jedoch schwerer, da die Anzahl der Nachbarn pro Punkt dort begrenzt ist.

Vorteil ist die symmetrische Natur der Nachbarschaftsbeziehungen. Weiterer Vorteil ist, dass nicht aufgrund einer festen Nachbaranzahl Kleinstcluster zusammengeschlossen werden.

Auf der anderen Seite muss aber ε geschickt initialisiert werden, um hier Erfolge zu erzielen, nämlich kleiner als die Hälfte des kleinsten Abstands zwischen zwei Clustern. Dies ist bei sehr variablen Clusterabständen und Punktabständen innerhalb von Clustern unter Umständen ein Problem.

Siehe für eine Veranschaulichung den unteren rechten Teil der Abb. A.1.

A.4 Der Silhouettenkoeffizient macht die Güte eines gegebenen Clusterings messbar

Wie wir oben sehen, gibt es keine Patentlösung für Clusteringprobleme, denn jedes dargestellte Verfahren hat seine ganz spezifischen Nachteile. Insofern ist es wertvoll, ein Kriterium dafür zu haben, wie gut unsere Clustereinteilung ist. Genau diese Möglichkeit bietet uns der ***Silhouettenkoeffizient*** nach [Kau90]. Er misst, wie gut die Cluster voneinander abgegrenzt sind, und ist ein Indikator dafür, ob vielleicht Punkte in falsche Cluster einsortiert sind.

Sei P eine Punktwolke und sei p ein Punkt $\in P$. Sei $c \subseteq P$ ein Cluster in der Punktwolke und gehöre p in diesen Cluster, also $p \in c$. Die Menge der Cluster nennen wir C. Zusammengefasst gilt also

$$p \in c \subseteq P.$$

Um den Silhouettenkoeffizient zu errechnen, benötigen wir zunächst den durchschnittlichen Abstand des Punktes p zu allen seinen Clusternachbarn. Diese Größe nennen wir $a(p)$ und definieren sie wie folgt:

$$a(p) = \frac{1}{|c| - 1} \sum_{q \in c, q \neq p} \operatorname{dist}(p, q) \tag{A.1}$$

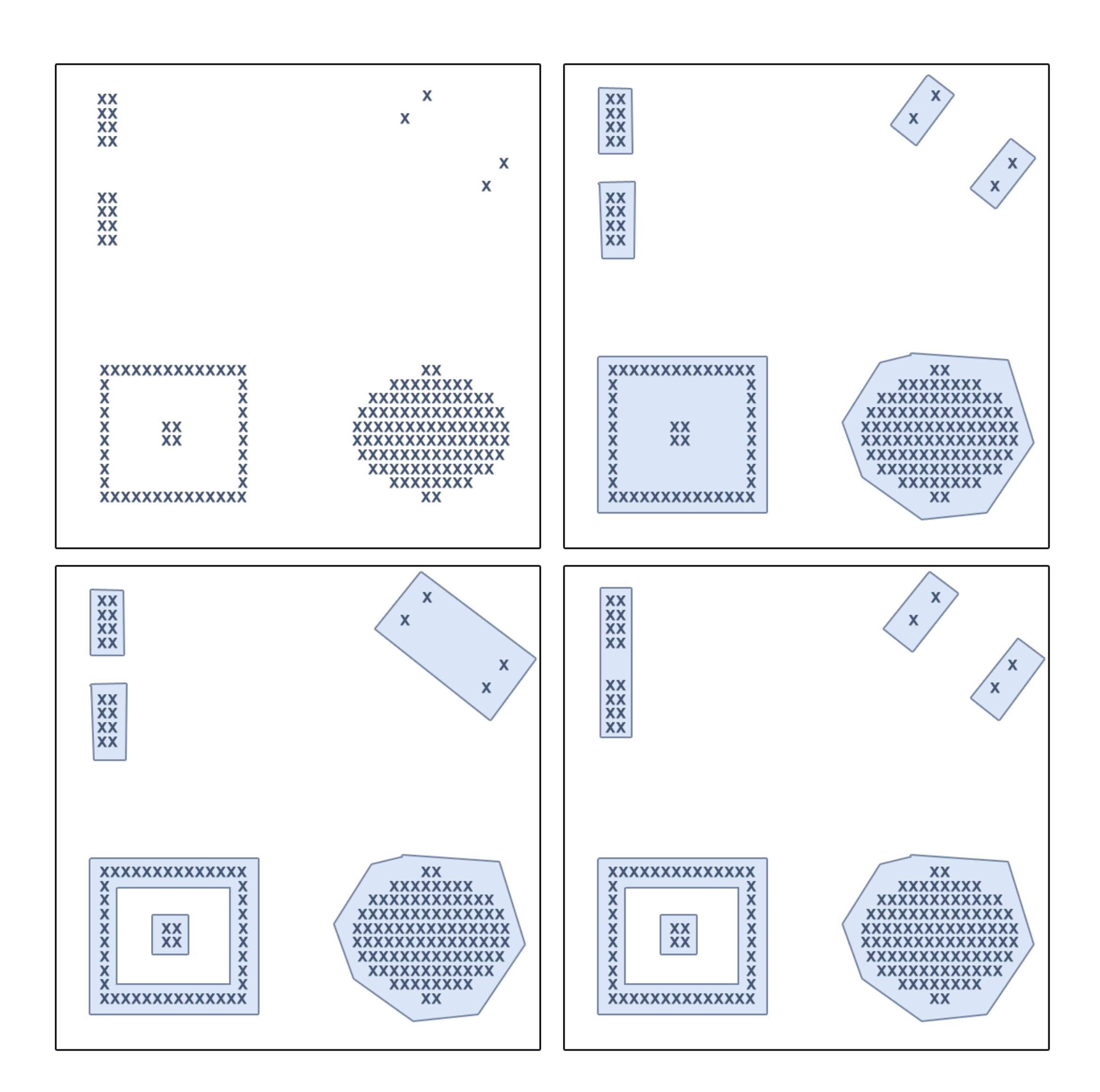


Abbildung A.1: Oben links: Unsere Punktmenge. An ihr werden wir die verschiedenen Clusteringverfahren erforschen. **Oben rechts: k-Means Clustering.** Bei der Anwendung dieses Verfahrens wurde $k = 6$ gewählt. Wie wir sehen, kann das Verfahren „Cluster in Clustern“ nicht erkennen (unten links im Bild). Auch lange „Linien“ von Punkten bereiten Schwierigkeiten: Sie würden als viele kleine Cluster erkannt werden (bei ausreichend großer Wahl von k). **Unten links: k-Nearest Neighbouring.** Bei zu großer Wahl von k (größer als die Punktanzahl des kleinsten Clusters) kommt es zu Clusterzusammenschlüssen, wie man oben rechts im Bild sehen kann. **Unten rechts: ε-Nearest Neighbouring.** Dieses Verfahren bereitet Schwierigkeiten, wenn ε größer gewählt ist als der minimale Abstand zwischen zwei Clustern (sichtbar oben links im Bild) – diese werden dann zusammengeschlossen.

Sei weiterhin $b(p)$ der durchschnittliche Abstand unseres Punktes p zu allen Punkten des nächsten anderen Clusters (g läuft über alle Cluster außer c):

$$b(p) = \min_{g \in C, g \neq c} \frac{1}{|g|} \sum_{q \in g} \text{dist}(p, q) \tag{A.2}$$

Der Punkt p ist gut klassifiziert, wenn der Abstand zum Schwerpunkt des eigenen Clusters minimal und der Abstand zu den Schwerpunkten der anderen Cluster maximal ist. Ist dies der Fall, so wird der folgende Term einen Wert nahe 1 ergeben:

$$s(p) = \frac{b(p) - a(p)}{\max\{a(p), b(p)\}} \tag{A.3}$$

Der ganze Term $s(p)$ kann sich offensichtlich nur im Intervall $[-1; 1]$ bewegen. Indikator für eine schlechte Klassifikation von p ist ein Wert nahe -1.

Der Silhouettenkoeffizient $S(P)$ ergibt sich aus dem Durchschnitte aller $s(p)$-Werte: Es gilt

$$S(P) = \frac{1}{|P|} \sum_{p \in P} s(p). \tag{A.4}$$

Die Gesamtgüte der Clustereinteilung drückt sich wie oben durch das Intervall $[-1; 1]$ aus.

Nachdem wir nun die Charakteristiken verschiedener Clusteringmethoden sowie ein Maß zur Qualitätsbeurteilung einer bereits existierenden Unterteilung in Cluster kennengelernt haben (viel weiteres Material findet sich in [DHS01]), möchte ich ein 2005 veröffentlichtes Clusteringverfahren auf Basis eines unüberwacht lernenden Neuronalen Netzes [SGE05] vorstellen, welches wie alle dieser Verfahren wahrscheinlich nicht perfekt ist, aber doch große Standard-Schwächen von den bekannten Clusteringverfahren ausmerzt – auch, um einmal eine vergleichsweise hochaktuelle Entwicklung im Bereich der Neuronalen Netze darzustellen.

A.5 Regional and Online Learnable Fields sind ein neuronales Clusteringverfahren

Das Paradigma Neuronaler Netze, welches ich nun vorstellen möchte, sind die ***Regional and Online Learnable Fields***, kurz ***ROLFs*** genannt.

A.5.1 ROLFs versuchen, mit Neuronen Datenwolken abzudecken

Grob gesagt sind die Regional and Online Learnable Fields eine Menge K von Neuronen, die versuchen, eine Menge von Punkten durch ihre Verteilung im Eingaberaum möglichst gut abzudecken. Hierfür werden während des Trainings bei Bedarf Neurone hinzugefügt, verschoben oder in ihrer Größe verändert. Die Parameter der einzelnen Neurone werden wir noch erforschen.

Definition A.2 (Regional and Online Learnable Field)**.** Ein Regional and Online Learnable Field (kurz: ROLF oder ROLF-Netz) ist eine Menge K von Neuronen, welche darauf trainiert werden, eine bestimmte Menge im Eingaberaum möglichst gut abzudecken.

A.5.1.1 ROLF-Neurone besitzen Position und Radius im Eingaberaum

Ein ***ROLF-Neuron*** $k \in K$ hat hierbei zwei Parameter: Es besitzt ähnlich der RBF-Netze ein ***Zentrum*** c_k, also einen Ort im Eingaberaum.

Es besitzt aber noch einen weiteren Parameter: Den Radius σ, der den Radius der ***perzeptiven Fläche***, welche das Neuron umgibt, erst definiert[2]. Ein Neuron deckt den Anteil des Eingaberaums ab, der innerhalb des Radius liegt.

Sowohl c_k als auch σ_k sind für jedes Neuron lokal definiert, das heißt insbesondere, dass die Neurone verschieden große Flächen abdecken können.

Der Radius der perzeptiven Fläche ist durch $r = \rho \cdot \sigma$ gegeben (Abb. A.2 auf der folgenden Seite), wobei der Multiplikator ρ für alle Neurone *global* definiert ist und im Vorhinein festgelegt wird. Der Leser wird sich nun intuitiv fragen, wozu dieser Multiplikator gut ist. Wir werden seine Bedeutung später noch erforschen. Zu beachten ist weiterhin: Die perzeptive Fläche der verschiedenen Neurone muss nicht gleich groß sein.

Definition A.3 (ROLF-Neuron)**.** Ein ROLF-Neuron k besitzt als Parameter ein Zentrum c_k sowie einen Radius σ_k.

Definition A.4 (Perzeptive Fläche)**.** Die perzeptive Fläche eines Rolf-Neurons k besteht aus allen Punkten, welche im Eingaberaum innerhalb des Radius $\rho \cdot \sigma$ liegen.

2 Ich schreibe daher „definiert“ und nicht „ist“, weil der eigentliche Radius ja durch $\sigma \cdot \rho$ gegeben ist.

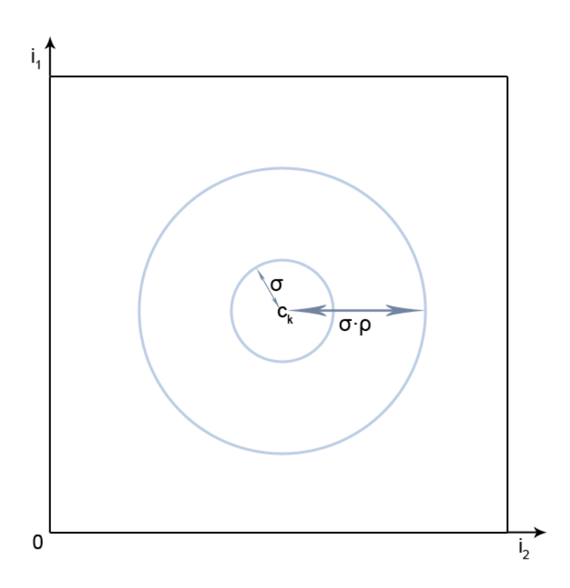


Abbildung A.2: Aufbau eines ROLF-Neurons.

A.5.2 ROLFs lernen unüberwacht durch Online-Präsentation von Trainingsbeispielen

Wie viele andere Paradigmen Neuronaler Netze lernt unser ROLF-Netz durch die Eingabe vieler Trainingsbeispiele p aus einer Trainingsmenge P. Das Lernen findet unüberwacht statt. Für jedes Trainingsbeispiel p, welches in das Netz eingegeben wird, können zwei Fälle eintreten:

1. Es existiert ein akzeptierendes Neuron k für p oder

2. es existiert kein akzeptierendes Neuron.

Falls im ersteren Fall mehrere Neurone in Frage kommen, existiert insofern *genau ein* ***akzeptierendes Neuron***, als dass das nächstgelegene akzeptierend ist. Beim akzeptierenden Neuron k werden c_k und σ_k angepasst.

Definition A.5 (Akzeptierendes Neuron)**.** Damit ein ROLF-Neuron k ein akzeptierendes Neuron eines Punktes p ist, muss der Punkt p innerhalb der perzeptiven Fläche von k liegen. Liegt p in den perzeptiven Flächen mehrerer Neurone, so ist das nächstliegende akzeptierend. Gibt es mehrere nächste Neurone, kann willkürlich eines ausgewählt werden.

A.5.2.1 Sowohl Positionen als auch Radien werden beim Lernen adaptiert

Angenommen, wir haben ein Trainingsbeispiel p in das Netz eingegeben und es existiert ein akzeptierendes Neuron k. Dann wandert der Radius in Richtung $||p - c_k||$ (also in Richtung des Abstandes zwischen p ud c_k) und das Zentrum c_k in Richtung von p. Zusätzlich seien zwei Lernraten η_σ und η_c für Radien und Zentren definiert.

$$c_k(t+1) = c_k(t) + \eta_c(p - c_k(t))$$
$$\sigma_k(t+1) = \sigma_k(t) + \eta_\sigma(||p - c_k(t)|| - \sigma_k(t))$$

Hierbei ist zu beachten, dass σ_k ein Skalar ist, während c_k ein Vektor im Eingaberaum ist.

Definition A.6 (Adaption eines ROLF-Neurons)**.** Ein zu einem Punkt p akzeptierendes Neuron k wird nach folgenden Regeln adaptiert:

$$c_k(t+1) = c_k(t) + \eta_c(p - c_k(t)) \tag{A.5}$$
$$\sigma_k(t+1) = \sigma_k(t) + \eta_\sigma(||p - c_k(t)|| - \sigma_k(t)) \tag{A.6}$$

A.5.2.2 Der Radiusmultiplikator sorgt dafür, dass Neurone nicht nur schrumpfen können

Nun können wir auch verstehen, wozu der Multiplikator ρ da ist: Durch ihn umfasst die perzeptive Fläche eines Neurons mehr als nur alle Punkte um das Neuron im Radius σ. Das bedeutet, dass σ durch die obige Lernregel nicht nur schrumpfen, sondern auch wachsen kann.

Definition A.7 (Radiusmultiplikator)**.** Der Radiusmultiplikator $\rho > 1$ ist global definiert und vergrößert die perzeptive Fläche eines Neurons k auf ein Vielfaches von σ_k. So ist sichergestellt, dass der Radius σ_k auch wachsen und nicht nur schrumpfen kann.

Der Radiusmultiplikator wird üblicherweise auf Werte im unteren einstelligen Bereich gesetzt, wie z.B. 2 oder 3.

Wir haben bis jetzt nur den Fall im ROLF-Training betrachtet, dass für ein Trainingsbeispiel p ein akzeptierendes Neuron existiert.

A.5.2.3 Nach Bedarf werden neue Neurone erzeugt

Dies legt nahe, die Vorgehensweise zu erforschen, falls kein akzeptierendes Neuron existiert.

Wenn dies der Fall ist, so wird einfach ein akzeptierendes Neuron k für unser Trainingsbeispiel *neu erzeugt*. Dies hat zur Folge, dass natürlich c_k und σ_k initialisiert werden müssen.

Intuitiv verstehbar ist die Initialisierung von c_k: Das Zentrum des neuen Neurons wird einfach auf das Trainingsbeispiel gesetzt, es gilt also

$$c_k = p.$$

Wir erzeugen ein neues Neuron, weil keins in der Nähe von p ist – also setzen wir das Neuron sinnvollerweise genau auf p.

Doch wie wird ein σ gesetzt, wenn ein neues Neuron gebildet wird? Hierfür gibt es verschiedene Taktiken:

Init-σ: Es wird immer ein vorherbestimmtes, statisches σ gewählt.

Minimum-σ: Wir schauen uns die σ aller Neurone an und wählen das Minimum.

Maximum-σ: Wir schauen uns die σ aller Neurone an und wählen das Maximum.

Mean-σ: Der Mittelwert der σ aller Neurone wird gewählt.

Aktuell ist die Mean-σ-Variante die favorisierte, obwohl das Lernverfahren mit den anderen auch funktioniert. Die Minimum-σ-Variante lässt die Neurone hierbei tendenziell weniger Fläche einnehmen, die Maximum-σ-Variante tendenziell mehr.

Definition A.8 (Erzeugen eines ROLF-Neurons)**.** Wird ein neues ROLF-Neuron k durch Eingabe eines Trainingsbeispiels p erzeugt, so wird c_k mit p initialisiert und σ_k nach einer der obigen Strategien (Init-σ, Minimum-σ, Maximum-σ, Mean-σ).

Ein gutes Kriterium für ein Trainingsende ist, wenn nach immer wieder zufällig permutierter Präsentation der Muster bei einer Epoche kein neues Neuron mehr erzeugt wurde und die Neuronenpositionen sich kaum noch verändern.

A.5.3 Auswertung eines ROLFs

Der Trainingsalgorithmus hat zur Folge, dass die ROLF-Neurone nach und nach die Trainingsmenge recht gut und genau abdecken, und dass eine hohe Konzentration von Punkten an einem Fleck des Eingaberaums nicht automatisch mehr Neurone erzeugen muss. Eine unter Umständen sehr große Punktwolke wird also auf (relativ zur Eingabemenge) sehr wenige Repräsentanten reduziert.

Die Anzahl der Cluster lässt sich dann sehr einfach bestimmen: Zwei Neurone sind (nach Definition beim ROLF) verbunden, wenn deren perzeptive Flächen sich überlappen (es wird also eine Art *Nearest Neighbouring* mit den variablen perzeptiven Flächen ausgeführt). Ein Cluster ist eine Gruppe von verbundenen Neuronen bzw. von diesen Neuronen abgedeckten Punkten des Eingaberaums (Abb. A.3 auf der folgenden Seite).

Selbstverständlich kann man das fertige ROLF-Netz auch mit anderen Clusteringverfahren auswerten, also Cluster in den Neuronen suchen. Insbesondere bei Clusteringverfahren, deren Speicheraufwand quadratisch zu $|P|$ steigt, kann der Speicheraufwand so dramatisch reduziert werden, da es in aller Regel wesentlich weniger ROLF-Neurone als ursprüngliche Datenpunkte gibt, die Neurone die Datenpunkte aber recht gut repräsentieren.

A.5.4 Vergleich mit populären Clusteringverfahren

Es ist offensichtlich, dass der Löwenanteil des Speicheraufwands bei den ROLFs beim Speichern der Neurone liegt und nicht etwa beim Speichern von Eingabepunkten. Dies ist von großem Vorteil bei riesigen Punktwolken mit sehr vielen Punkten.

Unser ROLF als neuronales Clusteringverfahren hat also, da die Datenpunkte nicht komplett gespeichert werden müssen, die Fähigkeit des *Online-Lernens*, was zweifellos einen großen Vorteil darstellt. Weiterhin kann es (ähnlich dem ε Nearest Neighbouring bzw. k Nearest Neighbouring) Cluster von eingeschlossenen Clustern unterscheiden – jedoch durch die Online-Präsentation der Daten ohne quadratischen Speicheraufwand, der mit Abstand der größte Nachteil der beiden Neighbouring-Verfahren ist.

Es wird weiterhin durch die variablen perzeptiven Flächen auf die Größe der jeweiligen Cluster im Verhältnis zu ihrer Entfernung voneinander eingegangen – was ebenfalls bei den beiden genannten Verfahren nicht immer der Fall ist.

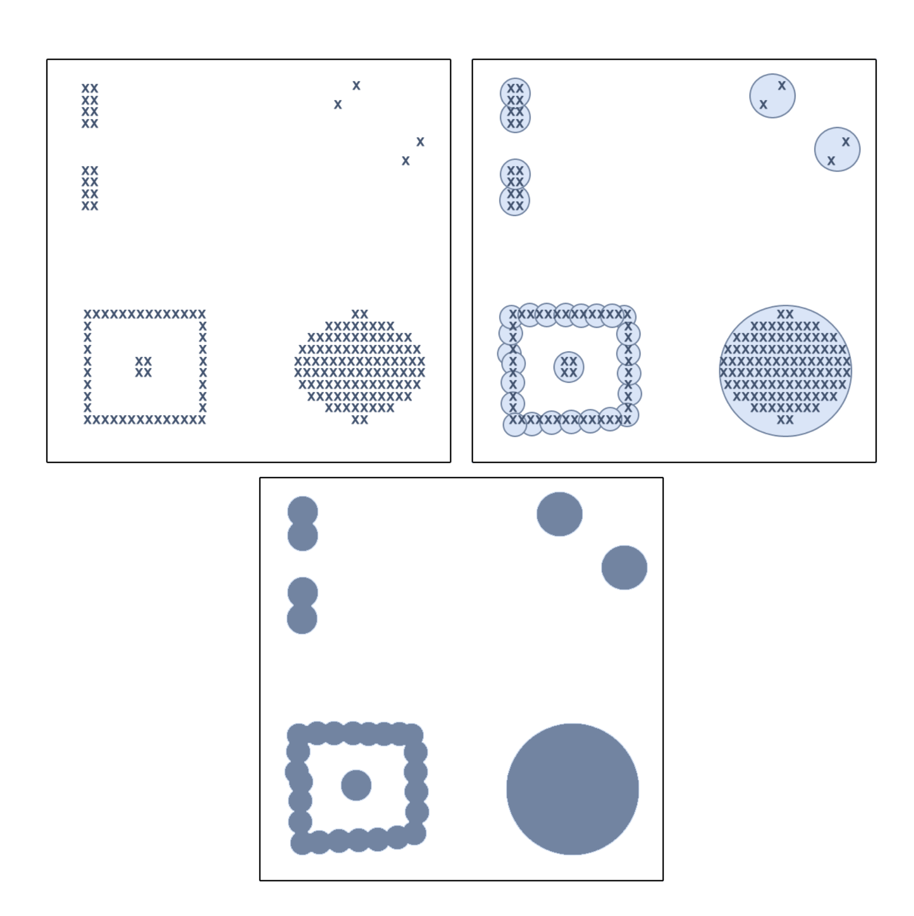


Abbildung A.3: Ablauf des Clusterings durch ein ROLF. Oben die Eingabemenge, in der Mitte die Abdeckung durch ROLF-Neurone, unten die reine Abdeckung durch die Neurone (Repräsentanten).

Im Vergleich mit dem k-Means-Clustering schneidet das ROLF auch gut ab: Erstens muss man nicht die Clusteranzahl im Vorhinein wissen, zum zweiten erkennt das k-Means-Clustering Cluster, die von anderen Clustern eingeschlossen sind, nicht als separate Cluster an.

A.5.5 Die Initialisierung von Radien, Lernraten und Multiplikator ist nicht trivial

Natürlich sollen auch die Nachteile des ROLFs nicht verschwiegen werden: Es ist nicht immer leicht, den Initialwert für die σ und das ρ gut zu wählen. In das ρ und den σ-Startwert kann man dem ROLF sozusagen das Vorwissen über die Datenmenge mitgeben: Feinkörnig geclusterte Daten sollten ein kleines ρ und einen kleinen σ-Startwert verwenden. Je kleiner aber das ρ, desto weniger Chance haben die Neuronen, größer zu werden, wenn nötig. Hier gibt es wieder keine Patentrezepte, genau wie für die Lernraten η_c und η_σ.

Beliebt für ρ sind Multiplikatoren im unteren einstelligen Bereich, wie 2 oder 3. Bei η_c und η_σ wird mit Werten um 0.005 bis 0.1 erfolgreich gearbeitet, auch bei diesem Netztyp sind Variationen zur Laufzeit denkbar. Startwerte für σ hängen in der Regel sehr von der Cluster- und Datenstreuung ab (müssen also oft ausprobiert werden), wobei diese gegenüber falschen Initialisierungen zumindest bei der Mean-σ-Strategie nach einiger Trainingszeit relativ robust sind.

Insgesamt muss sich das ROLF im Vergleich zu den anderen Clusteringverfahren durchaus nicht verstecken und wird insbesondere für speicherarme Systeme oder sehr große Datenmengen sehr interessant sein.

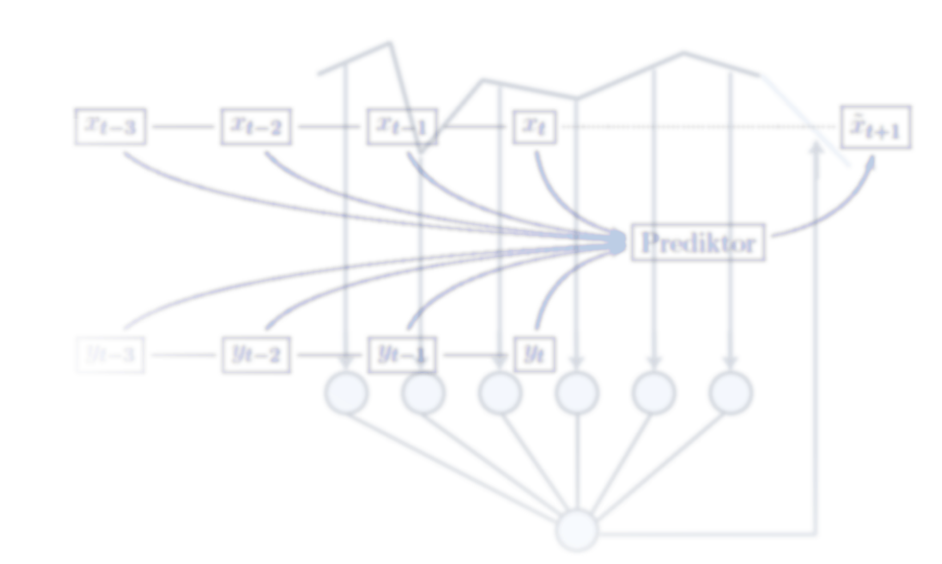
A.5.6 Anwendungsbeispiele

Ein erstes Anwendungsbeispiel könnte z.B. das Finden von Farbclustern in RGB-Bildern sein. Ein weiteres direkt in der ROLF-Veröffentlichung beschriebenes Einsatzgebiet ist das Erkennen von Worten, welche in einen 720dimensionalen Merkmalsraum überführt wurden – wir sehen also, dass die ROLFs auch gegenüber höheren Dimensionen relativ robust sind. Weitere mir bekannte Anwendungsgebiete liegen in der Entdeckung und Zuordnung von Angriffen auf Netzwerksysteme.

Übungsaufgaben

Aufgabe 19. Bestimmen Sie mindestens vier Adaptionsschritte für ein einzelnes ROLF-Neuron k, wenn die vier untenstehenden Muster nacheinander in der angegebenen Reihenfolge präsentiert werden. Die Startwerte für das ROLF-Neuron seien $c_k = (0.1, 0.1)$ und $\sigma_k = 1$. Weiter gelte $\eta_c = 0.5$ sowie $\eta_\sigma = 0$. Sei $\rho = 3$.

$$\begin{aligned} P = \{&(0.1, 0.1);\\ =&(0.9, 0.1);\\ =&(0.1, 0.9);\\ =&(0.9, 0.9)\}. \end{aligned}$$



Anhang B

Exkurs: Neuronale Netze zur Vorhersage

Betrachtung einer Anwendung Neuronaler Netze: Ein Blick in die Zukunft von Zeitreihen.

Nach Betrachtung verschiedenster Paradigmen Neuronaler Netze ist es sinnvoll, nun eine Anwendung Neuronaler Netze zu betrachten, die viel thematisiert und (wie wir noch sehen werden) auch für Betrug genutzt wird: Die Anwendung der ***Zeitreihenvorhersage***. Dieses Exkurskapitel gliedert sich hierbei in die Beschreibung von Zeitreihen und Abschätzungen, unter welchen Voraussetzungen man überhaupt nur Werte einer Zeitreihe vorhersagen kann. Zum Schluß möchte ich noch etwas zu den häufigen Softwareangeboten sagen, die mit Hilfe Neuronaler Netze oder anderer Verfahren Aktienkurse oder andere wirtschaftliche Kenngrößen vorhersagen sollen.

Das Kapitel soll weniger eine ausführliche Beschreibung sein, sondern vielmehr ein paar Denkansätze für die Zeitreihenvorhersage nennen, insofern werde ich mich mit formalen Definitionen wieder zurückhalten.

B.1 Über Zeitreihen

Eine ***Zeitreihe*** ist eine Reihe von Werten, welche in der Zeit diskretisiert ist. Beispielsweise könnten täglich gemessene Temperaturwerte oder andere Wetterdaten eines Ortes eine Zeitreihe darstellen – auch Aktienkurswerte stellen eine Zeitreihe dar. Zeitreihen sind häufig zeitlich äquidistant gemessen, und bei vielen Zeitreihen ist sehr von Interesse, wie denn die Zukunft ihrer Werte aussieht – nennen wir als Beispiel nur die tägliche Wettervorhersage.

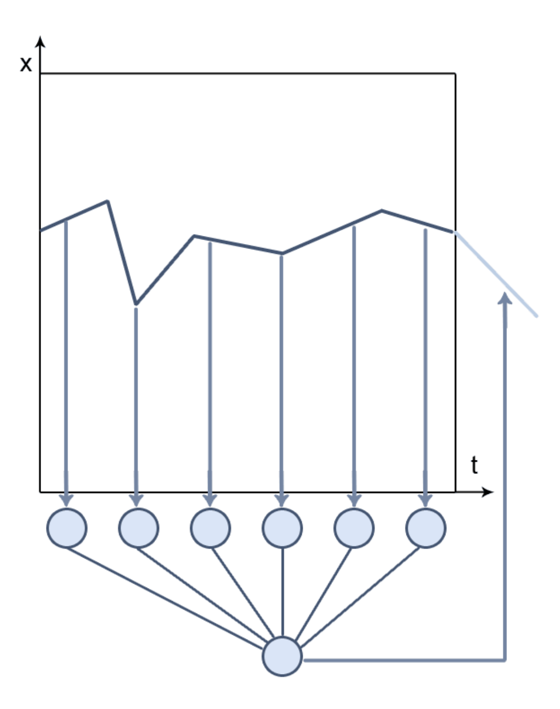


Abbildung B.1: Eine Funktion x der Zeit wird zu diskreten Zeitpunkten abgetastet (zeitdiskretisiert), wir erhalten also eine Zeitreihe. Die abgetasteten Werte werden in ein Neuronales Netz eingegeben (hier im Beispiel ein SLP), welches lernen soll, Werte der Zeitreihe vorherzusagen, welche in der Zukunft liegen.

Zeitreihen können auch in einem bestimmten zeitlichen Abstand Δt abgetastete Werte einer eigentlich kontinuierlichen Funktion sein (Abb. B.1).

Wenn wir eine Zeitreihe *vorhersagen* möchten, so suchen wir ein Neuronales Netz, was vergangene Werte der Zeitreihe auf zukünftige abbildet – kennen wir eine Zeitreihe also auf längeren Abschnitten, so sind uns genug Trainingsbeispiele gegeben. Diese sind natürlich keine Beispiele der vorherzusagenden Zukunft, aber man versucht mit ihrer Hilfe, die Vergangenheit zu verallgemeinern und zu extrapolieren.

Bevor wir uns allerdings an die Vorhersage einer Zeitreihe machen, müssen wir uns ein paar Gedanken zu Fragen über die betrachtete Zeitreihe machen – bzw. sicherstellen, dass unsere Zeitreihe einige Bedingungen erfüllt.

1. Haben wir überhaupt Anhaltspunkte dafür, dass die Zukunft der Zeitreihe auf irgendeine Weise von ihrer Vergangenheit abhängt? Steckt in der Vergangenheit der Zeitreihe also Information über ihre Zukunft?

2. Haben wir genug vergangene Werte der Zeitreihe als Trainingsmuster?

3. Im Falle einer Vorhersage einer kontinuierlichen Funktion: Wie müssen wir Δt sinnvollerweise wählen?

Wir wollen diese Fragen nun etwas näher beleuchten.

Wieviel Information über die Zukuft ist in der Vergangenheit einer Zeitreihe vorhanden? Dies ist mit Abstand die wichtigste Frage, die wir für jede Zeitreihe, die wir in die Zukunft abbilden wollen, beantworten müssen. Sind die zukünftigen Werte einer Zeitreihe beispielsweise überhaupt nicht von den vergangenen abhängig, so ist überhaupt keine Zeitreihenvorhersage aus ihnen möglich.

Wir gehen in diesem Kapitel von Systemen aus, deren Zustände auch auf ihre Zukunft schließen lassen – deterministische Systeme. Dies bringt uns erst einmal zu der Frage, was ein Systemzustand ist.

Ein Systemzustand beschreibt ein System für einen bestimmten Zeitpunkt *vollständig*. Die Zukunft eines deterministischen Systems wäre durch die vollständige Beschreibung seines aktuellen Zustands also eindeutig bestimmt.

Das Problem in der Realität ist, dass ein solcher Zustandsbegriff alle Dinge umfasst, die auf irgendeine Weise Einfluss auf unser System nehmen.

Im Falle unserer Wettervorhersage für einen Ort könnten wir die Temperatur, den Luftdruck und die Wolkendichte als den Wetterzustand des Ortes zu einem Zeitpunkt t durchaus bestimmen – doch der gesamte Zustand würde wesentlich mehr umfassen. Hier wären weltweite wettersteuernde Phänomene von Interesse, genauso wie vielleicht kleine lokale Phänomene, z.B. die Kühlanlage des örtlichen Kraftwerks.

Es ist also festzuhalten, dass der Systemzustand für die Vorhersage wünschenswert, aber nicht immer zu bekommen ist. Oft sind nur Teile des aktuellen Zustands erfassbar – wie beim Wetter die verschiedenen angesprochenen Wettergrößen.

Wir können aber diese Schwäche teilweise ausgleichen, indem wir nicht nur die beobachtbaren Teile eines einzigen (des letzten) Zustandes in die Vorhersage mit einfließen lassen, sondern mehrere vergangene Zeitpunkte betrachten. Hieraus wollen wir nun unser erstes Vorhersagesystem formen:

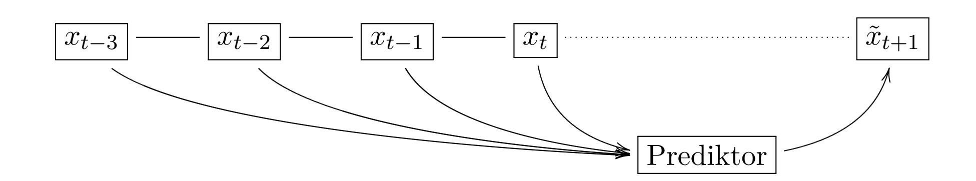


Abbildung B.2: Darstellung der One Step Ahead Prediction – aus einer Reihe vergangener Werte versucht man, den zukünftigen Wert zu errechnen. Das vorhersagende Element (in unserem Fall ein Neuronales Netz) nennt man Prediktor.

B.2 One Step Ahead Prediction

Den Vorhersageversuch für den nächsten zukünftigen Wert einer Zeitreihe aus vergangenen Werten nennen wir ***One Step Ahead Prediction*** (Abb. B.2).

Ein solches Prediktorsystem erhält also die letzten n observierten Zustandsteile des Systems als Eingabe und gibt die Vorhersage für den kommenden Zustand (oder Zustandsteil) aus. Die Idee, dass wir einen Zustandsraum mit Zuständen haben, die wir vorhersagen können, nennt man ***State Space Forecasting***.

Ziel des Prediktors ist es, eine Funktion

$$f(x_{t-n+1}, \dots, x_{t-1}, x_t) = \tilde{x}_{t+1} \tag{B.1}$$

zu realisieren, welche exakt n vergangene Zeitwerte entgegennimmt, um daraus den zukünftigen Wert vorherzusagen. Vorhergesagte Werte wollen wir mit einer Tilde (z.B. $\tilde{x}$) überschreiben, um sie von den tatsächlichen zukünftigen Werten zu unterscheiden.

Der intuitivste und einfachste Ansatz wäre, eine Linearkombination

$$\tilde{x}_{i+1} = a_0 x_i + a_1 x_{i-1} + \dots + a_j x_{i-j} \tag{B.2}$$

zu suchen, welche unsere Bedingungen näherungsweise erfüllt.

Eine solche Konstruktion nennt man ***digitales Filter***. Hier nutzt man aus, dass wir bei Zeitreihen in der Regel über sehr viele vergangene Werte verfügen, so dass wir eine Reihe von Gleichungen aufstellen können[1]:

$$
\begin{aligned}
x_t &= a_0 x_{t-1} + \ldots + a_j x_{t-1-(n-1)} \\
x_{t-1} &= a_0 x_{t-2} + \ldots + a_j x_{t-2-(n-1)} \\
&\vdots \\
x_{t-n} &= a_0 x_{t-n} + \ldots + a_j x_{t-n-(n-1)}
\end{aligned}
\tag{B.3}
$$

Man könnte also n Gleichungen für n viele unbekannte Koeffizienten finden und so auflösen (soweit möglich). Oder noch ein besserer Ansatz: Wir könnten $m > n$ Gleichungen für n Unbekannte so verwenden, dass die Summe der Fehlerquadrate der Vorhersagen, die uns ja bereits bekannt sind, minimiert wird – genannt ***Moving-Average-Verfahren***.

Diese lineare Struktur entspricht aber einfach einem Singlelayerperceptron mit linearer Aktivierungsfunktion, welches mit Daten aus der Vergangenheit trainiert wurde (Der Versuchsaufbau würde der Abb. B.1 auf Seite 226 entsprechen). In der Tat liefert ein Training mit der Deltaregel hier Ergebnisse, die der analytischen Lösung sehr nahe liegen.

Auch wenn man damit oft schon sehr weit kommt, haben wir gesehen, dass man mit einem Singlelayerperceptron viele Problemstellungen nicht abdecken kann. Weitere Schichten mit linearer Aktivierungsfunktion bringen uns auch nicht weiter, da ein Multilayerperceptron mit ausschließlich linearen Aktivierungsfunktionen sich auf ein Singlelayerperceptron reduzieren lässt. Diese Überlegungen führen uns zum nichtlinearen Ansatz.

Uns steht mit dem Multilayerperceptron und nichtlinearen Aktivierungsfunktionen ein universeller nichtlinearer Funktionsapproximator zur Verfügung, wir könnten also ein n-$|H|$-1-MLP für n viele Eingaben aus der Vergangenheit verwenden. Auch ein RBF-Netz wäre verwendbar, wir erinnern uns aber daran, dass die Zahl n hier im kleinen Rahmen bleiben muss, da hohe Eingabedimensionen in RBF-Netzen sehr aufwändig zu realisieren sind. Wenn man also viele Werte aus der Vergangenheit mit einbeziehen will, ist ein Multilayerperceptron deutlich weniger rechenaufwändig.

1 Ohne diesen Umstand weiter zu betrachten möchte ich anmerken, dass die Vorhersage oft einfacher wird, je mehr vergangene Werte der Zeitreihe zur Verfügung stehen. Ich bitte den Leser, hierzu etwas zum *Nyquist-Shannon-Abtasttheorem* zu recherchieren.

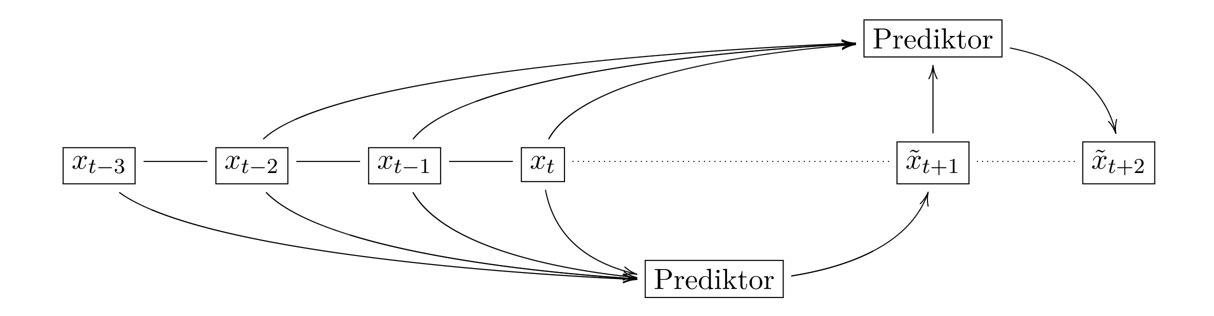


Abbildung B.3: Darstellung der Two Step Ahead Prediction – aus einer Reihe vergangener Werte versucht man, durch einen zweiten Prediktor unter Einbeziehung eines bereits vorhergesagten Wertes den zweiten zukünftigen Wert vorherzusagen.

B.3 Two Step Ahead Prediction

Was für Lösungsansätze können wir denn finden, wenn wir weiter in die Zukunft sehen wollen?

B.3.1 Rekursive Two Step Ahead Prediction

Um beispielsweise zwei Zeitschritte in die Zukunft zu schauen, könnten wir einfach zwei One Step Ahead Predictions hintereinander ausführen (Abb. B.3), also eine rekursive ***Two Step Ahead Prediction*** ausführen. Leider ist aber der von einer One Step Ahead Prediction ermittelte Wert in aller Regel nicht exakt, so dass sich die Fehler schnell aufschaukeln können und, je öfter man die Predictions hintereinander ausführt, das Ergebnis immer ungenauer wird.

B.3.2 Direkte Two Step Ahead Prediction

Wir ahnen schon, dass es eine bessere Variante gibt: So wie wir das System auf die Vorhersage des nächsten Wertes trainieren können, können wir das natürlich auch mit dem übernächsten Wert – wir trainieren also z.B. ein Neuronales Netz direkt darauf, zwei Zeitschritte in die Zukunft zu blicken, was wir ***Direct Two Step Ahead Prediction*** (Abb. B.4 auf der rechten Seite) nennen. Die Direct Two Step Ahead

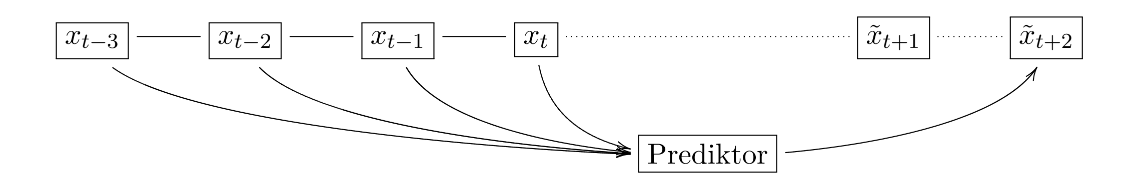


Abbildung B.4: Darstellung der Direct Two Step Ahead Prediction. Hier wird direkt der zweite Zeitschritt vorhergesagt, und der erste übersprungen. Sie unterscheidet sich technisch nicht von einer One Step Ahead Prediction.

Prediction ist offensichtlich technisch identisch mit der One Step Ahead Prediction, der Unterschied liegt nur im Training.

B.4 Weitere Optimierungsansätze für die Prediction

Die Möglichkeit, in der Zukunft weiter entfernt liegende Werte vorherzusagen, ist nicht nur wichtig, weil wir die Zukunft weiter vorhersagen zu versuchen – es kann auch periodische Zeitreihen geben, wo es anders schwer möglich ist: Wenn eine Vorlesung jeden Dienstag um 09:00 Uhr stattfindet, nützt uns die Information, wieviele Personen Montags im Hörsaal saßen, für die Prädiktion der Vorlesungsteilnehmerzahl sehr wenig. Gleiches gilt z.B. für periodisch auftretende Pendlerstaus.

B.4.1 Veränderung zeitlicher Parameter

Es kann also sehr sinnvoll sein, in der Zeitreihe sowohl in Vergangenheit als auch in Zukunft bewusst Lücken zu lassen, also den Parameter Δt einzuführen, der angibt, der wievielte zurückliegende Wert jeweils zur Prädiktion genutzt wird. Wir bleiben also technisch gesehen bei einer One Step Ahead Prediction, und strecken nur den Eingaberaum oder aber wir trainieren die Vorhersage weiter entfernt liegender Werte.

Auch Kombinationen verschiedener Δt sind möglich: Im Falle der Stauvorhersage für einen Montag könnten *zusätzlich* zu den letzten Montagen die letzten paar Tage als Dateninput genutzt werden. Wir nutzen so also die letzten Werte mehrerer Perioden, in

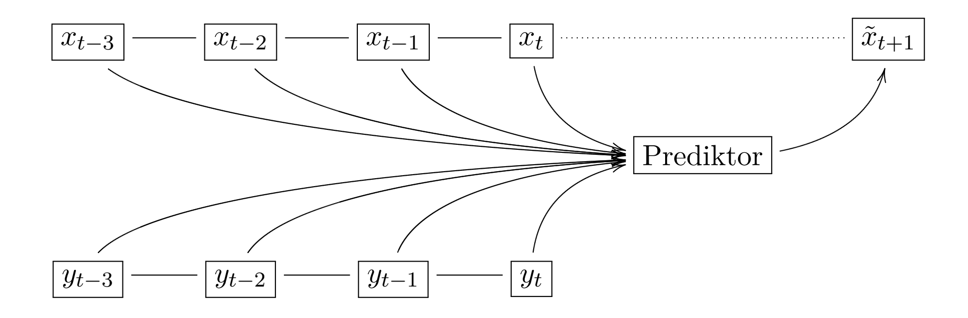


Abbildung B.5: Darstellung der Heterogenen One Step Ahead Prediction. Vorhersage einer Zeitreihe unter Betrachtung einer weiteren.

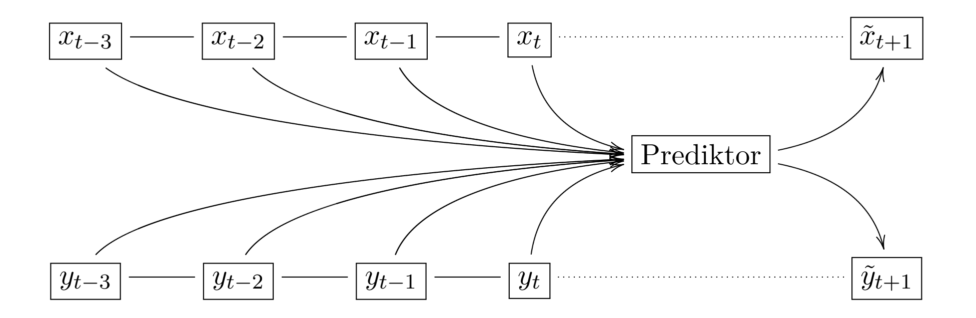


Abbildung B.6: Heterogene One Step Ahead Prediction von zwei Zeitreihen gleichzeitig.

diesem Fall einer wöchentlichen und einer täglichen. Wir könnten in Form der Ferienanfänge auch noch eine jährliche hinzunehmen (jeder von uns hat bestimmt schon einmal viel Zeit auf der Autobahn verbracht, weil er den Ferienanfang vergessen hatte).

B.4.2 Heterogene Prediction

Eine weitere Variante der Vorhersage wäre, aus mehreren Zeitreihen etwas über die Zukunft einer einzigen Zeitreihe vorherzusagen, falls man annimmt, dass die zusätzliche Zeitreihe etwas mit der Zukunft der ersten zu tun hat (***heterogene One Step Ahead Prediction***, Abb. B.5).

Will man zwei Ausgaben zu zwei Zeitreihen, welche etwas miteinander zu tun haben, vorhersagen, kann man natürlich zwei parallele One Step Ahead Predictions ausführen (dies wird analytisch oft gemacht, weil die Gleichungen sonst irgendwann sehr

unübersichtlich werden) – oder man hängt im Falle der Neuronalen Netze einfach ein Outputneuron mehr an und nutzt Wissen von beiden Zeitreihen für beide Ausgaben (Abb. B.6 auf der linken Seite).

Mehr und allgemeineres Material zum Thema „Zeitreihen“ findet sich unter [WG94].

B.5 Bemerkungen zur Vorhersage von Aktienkursen

Viele Personen schauen sich die Veränderung eines Aktienkurses in der Vergangenheit an und versuchen daraus auf die Zukunft zu schließen, um Profit daraus zu ziehen. Aktienkurse sind aus sich selbst heraus unstetig und daher prinzipiell schon einmal schwierige Funktionen. Weiterhin kann man die Funktionen nur zu diskreten Werten abrufen, oft beispielsweise im Tagesrhytmus (wenn man Glück hat, noch mit Maximal- und Minimalwerten pro Tag), wo natürlich die Tagesschwankungen wieder wegfallen. Dies macht die Sache auch nicht einfacher.

Es gibt nun *Chartisten*, also Personen, welche sich viele Diagramme anschauen und mit viel Hintergrundwissen und oft Jahrzehnten an Erfahrung entscheiden, ob Wertpapiere gekauft werden sollten oder nicht (und damit auch oft Erfolg haben).

Neben den Wertpapierkursen sind auch Wechselkurse von Währungen sehr interessant vorherzusagen: Wenn man 100 Euro in Dollar umtauscht, diese in Pfund und diese wieder in Euro, kann passieren, dass man zum Schluss 110 Euro herausbekommt. Wenn man das allerdings herausgefunden hat, würde man das öfter machen und somit selbst die Wechselkurse in einen Zustand verändern, in dem ein solcher vermehrender Kreislauf nicht mehr möglich ist (wäre das nicht so, könnte man ja Geld herstellen, indem man sozusagen ein finanzielles Perpetuum Mobile erschafft).

Gute Wertpapier- und Wechselkursbroker heben oder senken den Daumen an der Börse – und geben damit an, ob eine Aktie oder ein Wechselkurs ihrer Meinung nach steigen wird. Sie geben also mathematisch ausgedrückt das erste Bit (Vorzeichen) der ersten Ableitung des Wechselkurses an. Exzellente Weltklasse-Broker erreichen damit Erfolgsraten von etwas über 70%.

In Großbritannien ist es mit einer heterogenen One-Step-Ahead-Prediction gelungen, diese Vorhersagerichtigkeit auf 76% zu steigern: Zusätzlich zu der Zeitreihe des Wertes wurden noch *Indikatoren* miteinbezogen, wie z.B. der Ölpreis in Rotterdam oder die US-Staatsverschuldung.

Das einmal zur Größenordnung der Richtigkeit von Börsenschätzungen – wir reden ja immer noch über das Erste Bit der Ersten Ableitung! Damit ist uns auch noch keine

Information gegeben, wie stark der erwartete Anstieg oder Abfall ist, und so auch nicht, ob sich der ganze Aufwand lohnt: Vielleicht macht uns eine einzige falsche Vorhersage den ganzen Gewinn von hundert richtigen wieder zunichte.

Wie verhalten sich also Neuronale Netze zur Aktienkursvorhersage? Wir nehmen ja ganz intuitiv an, dass die Aktienkurswerte der Zukunft eine Funktion der Aktienwerte von Zeitpunkten davor sind.

Genau diese Annahme ist jedoch irrig: Aktienkurse sind keine Funktion ihrer Vergangenheit, sondern eine *Funktion ihrer mutmaßlichen Zukunft*. Wir kaufen keine Aktien, weil sie in den letzten Tagen sehr gestiegen sind – sondern weil wir *denken*, dass sie morgen höher steigen werden. Kaufen viele Personen aus diesem Einfall heraus eine Aktie, so treiben sie ihren Kurs in die Höhe, und haben also mit ihrer Mutmaßung recht – es entsteht eine ***Self Fulfilling Prophecy***, ein in der Wirtschaft lange bekanntes Phänomen.

Gleiches gilt für den umgekehrten Fall: Wir verkaufen Aktien, weil wir denken, dass sich *morgen* die Kurse nach unten bewegen werden – was den Kurs am nächten Tag nach unten drückt und am übernächsten in aller Regel noch viel mehr.

Immer wieder taucht Software auf, die mit wissenschaftlichen Schlagworten wie z.B. Neuronalen Netzen behauptet, Aktienkurse vorhersagen zu können. Kaufen Sie diese nicht – zusätzlich zu den oben genannten wissenschaftlichen Ausschlusskriterien aus einem einfachen Grund: Wenn diese Tools so toll funktionieren – warum verkauft die Herstellerfirma sie dann? Nützliches Wirtschaftswissen wird in aller Regel geheimgehalten, und wenn wir einen Weg wüssten, garantiert mit Aktien reich zu werden, würden wir doch durch diesen Weg Millionen verdienen, und nicht in 30-Euro-Häppchen durch den Verkauf desselben, oder?



Anhang C

Exkurs: Reinforcement Learning

Was, wenn keine Trainingsbeispiele existieren, man aber trotzdem beurteilen kann, wie gut man gelernt hat, ein Problem zu lösen? Betrachten wir ein Lernparadigma, welches zwischen überwachtem und unüberwachtem Lernen anzusiedeln ist.

Wir wollen nun einen eher exotischen Ansatz des Lernens kennenlernen – einfach, um einmal von den üblichen Verfahren wegzukommen. Wir kennen Lernverfahren, in denen wir einem Netz genau sagen, was es tun soll, also beispielhafte Ausgabewerte bereitstellen. Wir kennen ebenfalls Lernverfahren, wie bei den Self Organizing Maps, in denen ausschließlich Eingabewerte gegeben werden.

Wir wollen nun eine Art Mittelding erforschen: Das Lernparadigma des bestärkenden Lernens – *Reinforcement Learning* nach Sutton und Barto [SB98].

Reinforcement Learning an sich ist kein Neuronales Netz, sondern nur eines der drei Lernparadigmen, die wir bereits in Kapitel 4 genannt haben. Manche Quellen zählen es zu den überwachten Lernverfahren, da man ein Feedback gibt – durch die sehr rudimentäre Art des Feedbacks ist es aber begründet von den überwachten Lernverfahren abzugrenzen, mal ganz abgesehen von der Tatsache, dass es keine Trainingsbeispiele gibt.

Während allgemein bekannt ist, dass Verfahren wie Backpropagation im Gehirn selbst nicht funktionieren können, wird Reinforcement Learning allgemein als biologisch wesentlich motivierter angesehen.

Der Ausdruck ***Reinforcement Learning*** (***Bestärkendes Lernen***) kommt aus den Kognitionswissenschaften und der Psychologie und beschreibt das in der Natur überall vorhandene Lernsystem durch Zuckerbrot und Peitsche, durch gute Erfahrungen und

schlechte Erfahrungen, Belohnung und Bestrafung. Es fehlt aber eine Lernhilfe, die uns genau erklärt, was wir zu tun haben: Wir erhalten lediglich ein Gesamtergebnis für einen Vorgang (Haben wir das Schachspiel gewonnen oder nicht? Und wie sicher haben wir es gewonnen?), aber keine Ergebnisse für die Zwischenschritte.

Fahren wir beispielsweise mit unserem Fahrrad mit abgewetzten Reifen und einer Geschwindigkeit von exakt $21,5\frac{km}{h}$ in einer Kurve über Sand mit einer Korngröße von durchschnittlich 0.1mm, so wird uns niemand genau sagen können, auf welchen Winkel wir den Lenker einzustellen haben, oder noch schlimmer, wie stark die Muskelkontraktionen von unseren vielen Muskelteilen in Arm oder Bein dafür sein müssen. Je nachdem, ob wir das Ende der Kurve unbeschadet erreichen, sehen wir uns aber sehr schnell mit einer guten oder schlechten *Lernerfahrung*, einem Feedback bzw. *Reward* konfrontiert. Der Reward ist also sehr einfach gehalten – aber dafür auch wesentlich einfacher verfügbar. Wenn wir nun oft genug verschiedene Geschwindigkeiten und Kurvenwinkel ausgetestet haben und einige Rewards erhalten haben, werden wir in etwa ein Gefühl dafür bekommen, was funktioniert und was nicht: Genau dieses Gefühl zu erhalten, ist das Ziel des Reinforcement Learnings.

Ein weiteres Beispiel für die Quasi-Unmöglichkeit, eine Art Kosten- oder Nutzenfunktion zu erhalten, ist ein Tennisspieler, der durch komplizierte Bewegungen und ballistische Bahnen im dreidimensionalen Raum unter Einberechnung von Windrichtung, Wichtigkeit des Turniers, privaten Faktoren und vielem anderen versucht, seinen sportlichen Ruhm auf lange Zeit zu maximieren.

Um es gleich vorweg zu sagen: Da wir nur wenig Feedback erhalten, heißt Reinforcement Learning oft *ausprobieren* – und damit ist es recht langsam.

C.1 Systemaufbau

Wir wollen nun verschiedene Größen und Bestandteile des Systems kurz ansprechen, und sie in den nachfolgenden Abschnitten genauer definieren. Reinforcement Learning repräsentiert grob formuliert die gegenseitige Interaktion zwischen einem *Agenten* und einem *Umweltsystem* (Abb. C.2).

Der Agent soll nun irgendeine Aufgabe lösen, er könnte z.B. ein autonomer Roboter sein, der Hindernisvermeidung betreiben soll. Der Agent führt in der Umwelt nun Aktionen aus und bekommt von der Umwelt dafür ein Feedback zurück, das wir im folgenden *Reward* nennen wollen. Dieser Kreis aus Aktion und Reward ist charakteristisch für Reinforcement Learning. Der Agent beeinflusst das System, das System gibt einen Reward und verändert sich.

Der Reward ist ein reeller oder diskreter Skalar, welcher uns wie oben beschrieben angibt, wie gut wir unser Ziel erreichen, jedoch keine Anleitung vermittelt, *wie* wir es erreichen können. Ziel ist immer, langfristig eine möglichst hohe Summe von Rewards zu erwirtschaften.

C.1.1 Die Gridworld

Als Lernbeispiel für Reinforcement Learning möchte ich gerne die sogenannte ***Gridworld*** verwenden. Wir werden sehen, dass sie sehr einfach aufgebaut und durchschaubar ist und daher eigentlich gar kein Reinforcement Learning notwendig ist – trotzdem eignet sie sich sehr gut, die Vorgehensweisen des Reinforcement Learning an ihr darzustellen. Definieren wir nun die einzelnen Bestandteile des Reinforcement Systems beispielhaft durch die Gridworld. Wir werden jedes dieser Bestandteile später noch genauer beleuchten.

Umwelt: Die Gridworld (Abb. C.1 auf der folgenden Seite) ist eine einfache, diskrete Welt in zwei Dimensionen, die wir im Folgenden als *Umweltsystem* verwenden wollen.

Agent: Als *Agent* nehmen wir einen einfachen Roboter, der sich in unserer Gridworld befindet.

Zustandsraum: Wie wir sehen, hat unsere Gridworld 5×7 Felder, von denen 6 nicht begehbar sind. Unser Agent kann also 29 Positionen in der Gridworld besetzen. Diese Positionen nehmen wir für den Agenten als *Zustände*.

Aktionsraum: Fehlen noch die *Aktionen*. Definieren wir einfach, der Roboter könnte jeweils ein Feld nach oben, unten, rechts oder links gehen (solange dort kein Hindernis oder der Rand unserer Gridworld ist).

Aufgabe: Die Aufgabe unseres Agenten ist es, aus der Gridworld hinauszufinden. Der Ausgang befindet sich rechts von dem hell ausgefüllten Feld.

Nichtdeterminismus: Die beiden Hindernisse können durch eine „Tür“ verbunden werden. Wenn die Tür geschlossen ist (unterer Teil der Abbildung), ist das entsprechende Feld nicht begehbar. Die Tür kann sich nicht während eines Durchlaufs verändern, sondern nur zwischen den Durchläufen.

Wir haben nun eine kleine Welt geschaffen, die uns über die nachfolgenden Lernstrategien begleiten und sie uns anschaulich machen wird.

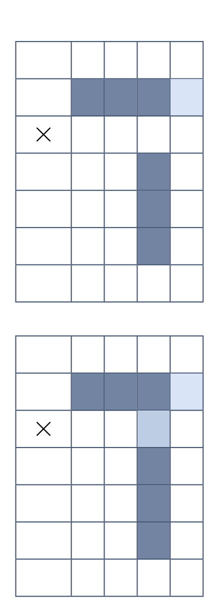


Abbildung C.1: Eine graphische Darstellung unserer Gridworld. Dunkel gefüllte Zellen sind Hindernisse und daher nicht begehbar. Rechts von dem hell gefüllten Feld ist der Ausgang. Das Symbol $\times$ markiert die Startposition unseres Agenten. Im oberen Teil der Abbildung ist die Tür offen, im unteren geschlossen.

C.1.2 Agent und Umwelt

Unser Ziel ist nun, dass der Agent *lernt*, was mit Hilfe des Rewards geschieht. Es wird also über, von und mit einem dynamischen System, der ***Umwelt***, gelernt, um ein Ziel zu erreichen. Doch was genau heißt eigentlich Lernen in diesem Zusammenhang?

Der ***Agent*** soll eine *Abbildung von Situationen auf Aktionen* (genannt *Policy*) lernen, also lernen, was er in welcher Situation tun soll, um ein ganz bestimmtes (gegebenes) Ziel zu erreichen. Das Ziel wird dem Agenten einfach aufgezeigt, indem er für das Erreichen eine Belohnung bekommt.

Die Belohnung ist nicht zu verwechseln mit dem Reward – vielleicht ist es auf dem Weg des Agenten zur Problemlösung auch sinnvoll, zwischendurch hin und wieder etwas weniger Belohnung oder gar Strafe zu bekommen, wenn das langfristige Ergebnis dafür maximal ist (ähnlich, wie wenn ein Anleger ein Tief eines Aktienkurses einfach aussitzt oder ein Bauernopfer beim Schach). Ist der Agent also auf einem guten Weg zum Ziel, gibt es positiven Reward, wenn nicht, gibt es keinen oder sogar negativen Reward

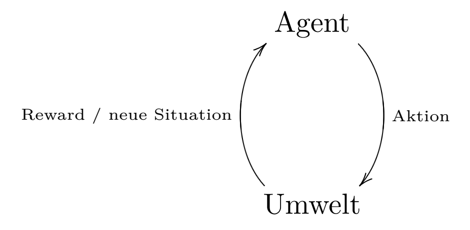


Abbildung C.2: Der Agent führt Aktionen in seiner Umwelt durch, welche ihm einen Reward gibt.

(Strafe). Die Belohnung ist sozusagen die schlussendliche Summe aller Rewards – wir wollen sie auch *Return* nennen.

Nachdem wir nun die Grundbestandteile umgangssprachlich benannt haben, wollen wir in den folgenden Abschnitten genauer betrachten, aus was wir unser Reinforcement-Learning-System abstrakt zusammensetzen können.

In der Gridworld: Der Agent ist in der Gridworld ein einfacher Roboter, der aus der Gridworld herausfinden soll. Umwelt ist die Gridworld selbst, eine diskrete Gitterwelt.

Definition C.1 (Agent)**.** Der Agent bei Reinforcement Learning kann formal beschrieben werden als eine Abbildung vom Situationsraum S in den Aktionsraum $A(s_t)$. Was Situationen s_t sind, wird später noch definiert und soll nur aussagen, dass der Aktionsraum von der aktuellen Situation abhängig ist.

$$\text{Agent: } S \to A(s_t) \tag{C.1}$$

Definition C.2 (Umwelt)**.** Die Umwelt repräsentiert eine stochastische Abbildung von einer Aktion A unter der aktuellen Situation s_t auf einen Reward r_t und eine neue Situation s_{t+1}.

$$\text{Umwelt: } S \times A \to P(S \times r_t) \tag{C.2}$$

C.1.3 Zustände, Situationen und Aktionen

Wie wir schon angesprochen haben, kann ein Agent sich innerhalb seiner Umwelt in verschiedenen ***Zuständen*** befinden: Im Falle der Gridworld zum Beispiel an verschiedenen Orten (wir erhalten hier einen zweidimensionalen Zustandsvektor).

Es ist für den Agenten aber nicht immer möglich, alle Informationen seines aktuellen Zustandes zu erfassen, weswegen wir den Begriff der ***Situation*** einführen müssen. Eine Situation ist ein Zustand *aus Agentensicht*, also nur eine mehr oder weniger gute *Approximation eines Zustandes.*

Situationen lassen es daher nicht im Allgemeinen zu, Folgesituationen eindeutig „vorherzusagen" – selbst bei einem vollständig deterministischen System haben wir das vielleicht nicht gegeben. Wenn wir alle Zustände und die Übergänge dazwischen exakt kennen würden (also das gesamte System), wäre eine optimale Planung möglich und auch eine optimale Policy einfach findbar – (Methoden liefert z.B. die dynamische Programmierung).

Wir wissen nun, dass das Reinforcement Learning eine Interaktion zwischen Agent und System mit ***Aktionen*** a_t und Situationen s_t ist. Der Agent kann nun nicht selbst feststellen, ob die aktuelle Situation gut oder schlecht ist: Genau dies ist der Grund, warum er wie eingangs bereits beschrieben von der Umwelt einen Reward erhält.

In der Gridworld: Zustände sind die Orte, an denen der Agent sich befinden kann. Situationen kommen den Zuständen in der Gridworld vereinfachend gleich. Mögliche Aktionen sind nach Norden, Süden, Osten oder Westen zu gehen.

Situation und Aktion können vektoriell sein, der Reward jedoch ist immer ein Skalar (im Extremfall sogar nur ein Binärwert), da das Ziel von Reinforcement Learning ist, mit sehr wenig Feedback auszukommen – ein komplizierter vektorieller Reward käme ja einem richtigen Teaching Input gleich.

Im Übrigen soll ja eine Kostenfunktion minimiert werden, was aber mit einem vektoriellen Reward so nicht möglich wäre, da wir keine intuitiven Ordnungsrelationen im Mehrdimensionalen besitzen – also nicht direkt wissen, was jetzt besser oder schlechter ist.

Definition C.3 (Zustand)**.** In einem Zustand befindet sich der Agent innerhalb seiner Umwelt. Zustände enthalten jede Information über den Agent im Umweltsystem. Es ist also theoretisch möglich, aus diesem gottähnlichen Zustandswissen einen Folgezustand auf eine ausgeführte Aktion in einem deterministischen System eindeutig vorherzusagen.

Definition C.4 (Situation)**.** Situationen s_t (hier zum Zeitpunkt t) aus einem ***Situationsraum*** S sind das eingeschränkte, approximative Wissen des Agenten über seinen Zustand. Die Approximation (von der der Agent nicht einmal wissen kann, wie gut sie ist) macht eindeutige Vorhersagen unmöglich.

Definition C.5 (Aktion)**.** Aktionen a_t können vom Agenten ausgeführt werden (wobei es sein kann, dass je nach Situation ein anderer ***Aktionsraum*** $A(S)$ besteht) und bewirken Zustandsübergänge und damit eine neue Situation aus Sicht des Agenten.

C.1.4 Reward und Return

Wie im wirklichen Leben ist unser Ziel, eine möglichst hohe Belohnung zu erhalten, also die Summe der erwarteten ***Rewards*** r, genannt ***Return*** R, langfristig zu maximieren. Bei endlich vielen Zeitschritten[1] kann man die Rewards einfach aufsummieren:

$$R_t = r_{t+1} + r_{t+2} + \dots \tag{C.3}$$

$$= \sum_{x=1}^{\infty} r_{t+x} \tag{C.4}$$

Der Return wird hierbei natürlich nur abgeschätzt (Würden wir alle Rewards und damit den Return komplett kennen, bräuchten wir ja nicht mehr lernen).

Definition C.6 (Reward)**.** Ein Reward r_t ist eine skalare, reelle oder diskrete (manchmal sogar nur binäre) Belohnungs- oder Bestrafungsgröße, welche dem Agenten vom Umweltsystem als Reaktion auf eine Aktion gegeben wird.

Definition C.7 (Return)**.** Der Return R_t ist die Aufkumulierung aller erhaltenen Rewards bis zum Zeitpunkt t.

C.1.4.1 Umgang mit großen Zeiträumen

Nicht alle Problemstellungen haben aber ein explizites Ziel und damit eine endliche Summe (unser Agent kann zum Beispiel ein Roboter sein, der die Aufgabe hat, einfach immer weiter herumzufahren und Hindernissen auszuweichen).

Um im Falle einer unendlichen Reihe von Reward-Abschätzungen keine divergierende Summe zu erhalten, wird ein abschwächender Faktor $0 < \gamma < 1$ verwendet, der den Einfluss ferner erwarteter Rewards abschwächt: Das ist nicht nur dann sinnvoll, wenn kein *Ziel* an sich existiert, sondern auch, wenn das Ziel sehr weit entfernt ist:

$$R_t = r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots \tag{C.5}$$

$$= \sum_{x=1}^{\infty} \gamma^{x-1} r_{t+x} \tag{C.6}$$

1 Soviel sind in der Praxis ja nur möglich, auch wenn die Formeln prinzipiell mit unendlichen Summen arbeiten

Je weiter die Belohnung weg ist, umso weniger Anteil hat sie also an den Entscheidungen des Agenten.

Eine andere Möglichkeit, die Return-Summe zu handhaben, wäre ein begrenzter ***Zeithorizont*** τ, so dass nur τ viele folgende Rewards $r_{t+1}, \ldots, r_{t+\tau}$ betrachtet werden:

$$R_t = r_{t+1} + \ldots + \gamma^{\tau-1} r_{t+\tau} \tag{C.7}$$

$$= \sum_{x=1}^{\tau} \gamma^{x-1} r_{t+x} \tag{C.8}$$

Wir unterteilen also den Zeitstrahl in ***Episoden***. Üblicherweise wird eine der beiden Methoden zur Begrenzung der Summe verwendet, wenn nicht sogar beide gemeinsam.

Wir versuchen also wie im täglichen Leben, unsere aktuelle Situation an einen *gewünschten Zustand* zu approximieren. Da nicht zwangsläufig nur der folgende erwartete Reward, sondern die erwartete *Gesamtsumme* bestimmt, was der Agent tut, können auch Aktionen durchgeführt werden, die kurzfristig erst einmal negativen Reward ergeben (z.B. das Bauernopfer beim Schach), sich jedoch später auszahlen.

C.1.5 Die Policy

Nachdem wir nun einige Systemelemente des Reinforcement Learnings genau betrachtet und formalisiert haben, bleibt noch das eigentliche Ziel zu betrachten:

Der Agent lernt während des Reinforcement Learnings eine ***Policy***

$$\Pi : S \to P(A),$$

er justiert also fortlaufend eine Abbildung von den Situationen auf die Wahrscheinlichkeiten $P(A)$, mit denen jede Aktion A in jeder Situation S ausgeführt wird. Eine Policy kann definiert werden als eine *Strategie, Aktionen auszuwählen, die den Reward auf lange Zeit maximiert.*

In der Gridworld: Die Policy ist in der Gridworld die Strategie, nach der der Agent versucht, aus der Gridworld herauszufinden.

Definition C.8 (Policy)**.** Die Policy Π ist eine Abbildung von Situationen auf Wahrscheinlichkeiten, jede Aktion aus dem Aktionsraum A auszuführen. Sie ist also formalisierbar als

$$\Pi : S \to P(A). \tag{C.9}$$

Wir unterscheiden hierbei grundsätzlich zwischen zwei Paradigmen von Policies: Eine ***Open Loop Policy*** stellt eine offene Steuerkette dar und bildet aus einer Startsituation s_0 eine Sequenz von Aktionen $a_0, a_1, \ldots$ mit $a_i \neq a_i(s_i); i > 0$. Der Agent erstellt also zu Beginn einen Plan und führt ihn sukzessive bis zum Ende aus, ohne die zwischenzeitlichen Situationen zu berücksichtigen (daher $a_i \neq a_i(s_i)$, Aktionen nach a_0 hängen nicht von den Situationen ab).

In der Gridworld: Eine Open Loop Policy würde uns in der Gridworld für eine Startposition eine exakte Wegbeschreibung zum Ausgang liefern, z.B. von der angegebenen Startposition den Weg (in Himmelsrichtungsabkürzungen) `OOOON`.

Eine Open Loop Policy ist also eine Sequenz von Aktionen ohne zwischenzeitliches Feedback. Aus einer Startsituation wird eine Sequenz von Aktionen generiert. Wenn man das System 100%ig kennt, kann man mit einer solchen Open Loop Policy erfolgreich arbeiten und zu sinnvollen Ergebnissen gelangen. Um aber z.B. das Schachspiel 100% zu kennen, müssten wir alle Spielzüge durchprobieren, was sehr aufwändig ist. Wir müssen uns für derartige Problemstellungen also eine Alternative zur Open Loop Policy suchen, die aktuelle Situationen in die Handlungsplanung miteinbezieht:

Eine ***Closed Loop Policy*** ist ein geschlossener Regelkreis, gewissermaßen eine Funktion

$$\Pi : s_i \to a_i \text{ mit } a_i = a_i(s_i).$$

Hier übt die Umwelt Einfluss auf unsere Aktionen aus bzw. der Agent reagiert auf Input der Umwelt, wie schon in Abb. C.2 dargestellt. Die Closed Loop Policy ist gewissermaßen ein reaktiver Plan, der aktuelle Situationen auf auszuführende Aktionen abbildet.

In der Gridworld: Eine Closed Loop Policy würde auf die Aktuelle Position eingehen und Richtungen nach der Aktion auswählen. Insbesondere wenn dynamisch ein Hindernis erscheint, ist eine solche Policy die bessere Wahl.

Bei der Wahl der auszuführenden Aktionen können wieder zwei grundsätzliche Strategien betrachtet werden.

C.1.5.1 Exploitation vs. Exploration

Wie im wirklichen Leben stellt sich während des Reinforcement Learnings oft die Frage, ob vorhandenes Wissen stur ausgenutzt, oder aber auch neue Wege ausprobiert werden. Wir wollen zunächst die zwei Extrema betrachten:

Eine ***Greedy Policy*** wählt immer den Weg des höchsten Rewards, den wir im voraus bestimmen können, also des höchsten bekannten Rewards. Diese Policy repräsentiert

den ***Exploitation-Ansatz*** und ist erfolgversprechend, wenn man das behandelte System bereits kennt.

Im Gegensatz zum Exploitation-Ansatz steht der ***Exploration-Ansatz***, der zum Ziel hat, ein System möglichst umfassend zu erforschen, so dass auch Wege zum Ziel gefunden werden können, welche auf den ersten Blick vielleicht nicht erfolgversprechend aussehen, es aber dennoch sind.

Angenommen, wir suchen einen Weg zum Restaurant, so wäre eine auf Nummer sicher gehende Policy, von überall den Weg zu nehmen, den wir kennen, so unoptimal und lang er auch sein mag, und nicht zu versuchen, bessere Wege zu erforschen. Ein anderer Ansatz wäre, auch hin und wieder nach kürzeren Wegen zu forschen, selbst auf die Gefahr hin, dass die Forschung lange dauert, nichts bringt und wir daher zum Schluß doch den ursprünglichen Weg nehmen und zu spät ins Restaurant kommen.

In der Realität wird oft eine Kombination beider Verfahren angewandt: Zum Anfang eines Lernvorgangs wird mit höherer Wahrscheinlichkeit geforscht, während zum Ende mehr vorhandenes Wissen ausgenutzt wird. Auch eine statische Wahrscheinlichkeitsverteilung ist hier möglich und wird oft angewandt.

In der Gridworld: Für die Wegsuche in der Gridworld gilt das Restaurantbeispiel natürlich analog.

C.2 Lernvorgang

Betrachten wir wieder das tägliche Leben. Von einer Situation können wir durch Aktionen in verschiedene Untersituationen geraten, von jeder Untersituation wieder in Unteruntersituationen. Gewissermaßen erhalten wir einen ***Situationsbaum***, wobei man Verknüpfungen unter den Knoten berücksichtigen muss (oft gibt es mehrere Wege, wie man zu einer Situation gelangen kann – der Baum könnte also treffender als *Situationsgraph* bezeichnet werden). Blätter des Baums sind Endsituationen des Systems. Der Exploration-Ansatz würde den Baum möglichst genau durchsuchen und alle Blätter kennenlernen, der Exploitation-Ansatz zielsicher zum besten bekannten Blatt gehen.

Analog zum Situationsbaum können wir uns auch einen Aktionsbaum aufbauen – hier stehen dann in den Knoten die Rewards für die Aktionen. Wir müssen nun vom täglichen Leben adaptieren, wie genau wir lernen.

C.2.1 Strategien zur Rewardvergabe

Interessant und von großer Wichtigkeit ist die Frage, wofür man einen Reward vergibt und was für einen Reward man vergibt, da das Design des Rewards das Systemverhalten maßgeblich steuert. Wie wir oben gesehen haben, gibt es (wieder analog zum täglichen Leben) in der Regel zu jeder Situation verschiedene Aktionen, welche man ausführen kann. Es existieren verschiedene Strategien, um die ausgewählten Situationen zu bewerten und so die zum Ziel führende Aktionsfolge zu lernen. Diese sollen im Folgenden erst einmal grundsätzlich erläutert werden.

Wir wollen nun einige Extremfälle als Designbeispiele zum Reward anbringen:

Als ***Pure Delayed Reward*** bezeichnen wir eine Rewardvergabe ähnlich wie beim Schachspiel: Wir erhalten unsere Belohnung zum Schluss, und während des Spiels keine. Diese Methode ist immer dann von Vorteil, wenn man am Ende genau sagen kann, ob man Erfolg gehabt hat, aber in den Zwischenschritten nicht genau einschätzen kann, wie gut die eigene Situation ist. Es gilt

$$r_t = 0 \; \forall t < \tau \tag{C.10}$$

sowie $r_\tau = 1$ bei Gewinn und $r_\tau = -1$ bei Verlust. Bei dieser Rewardstrategie geben nur die Blätter des Situationsbaumes einen Reward zurück.

Pure Negative Reward: Hier gilt

$$r_t = -1 \; \forall t < \tau. \tag{C.11}$$

Dieses System findet den schnellsten Weg zum Ziel, weil dieser automatisch auch der günstigste in Bezug auf den Reward ist. Man wird bestraft für alles, was man tut – selbst wenn man nichts tut, wird man bestraft. Diese Strategie hat zur Folge, dass es die preiswerteste Methode für den Agenten ist, schnell fertig zu werden.

Als weitere Strategie betrachten wir die ***Avoidance Strategy***: Schädlichen Situationen wird aus dem Weg gegangen. Es gilt

$$r_t \in \{0, -1\}, \tag{C.12}$$

Fast alle Situationen erhalten hier gar keinen Reward, nur einige wenige erhalten einen negativen. Diese negativen Situationen wird der Agent weiträumig umgehen.

Achtung: Rewardstrategien können leicht unerwartete Folgen haben. Ein Roboter, dem man sagt „mach was du willst, aber wenn du ein Hindernis berührst, kriegst du eine Strafe“, wird einfach stehenbleiben. Wird Stehenbleiben folglich auch bestraft, wird er kleine Kreise fahren. Bei näherem Überlegen kommt man auf den Gedanken, dass diese

Verhaltensweisen den Return des Roboters optimal erfüllen, aber leider nicht von uns intendiert waren.

Man kann weiterhin zeigen, dass insbesondere kleine Aufgaben mit negativen Rewards besser gelöst werden können während man bei großen, komplizierten Aufgaben mit positiven, differenzierteren Rewards weiter kommt.

In Bezug auf unsere Gridworld wollen wir den Pure Negative Reward als Strategie wählen: Der Roboter soll möglichst schnell zum Ausgang finden.

C.2.2 Die State-Value-Funktion

Im Gegensatz zu unserem Agenten haben wir eine gottgleiche Sicht auf unsere Gridworld, so dass wir schnell bestimmen können, welche Roboterstartposition welchen optimalen Return erreichen kann.

In Abbildung C.3 auf der rechten Seite sind diese optimalen Returns pro Feld aufgetragen.

In der Gridworld: Die State-Value-Funktion für unsere Gridworld stellt genau eine solche Funktion pro Situation (= Ort) dar, mit dem Unterschied, dass sie nicht bekannt ist, sondern gelernt werden muss.

Wir sehen also, dass es für den Roboter praktisch wäre, die aktuellen wie zukünftigen Situationen einer *Bewertung* unterziehen zu können. Betrachten wir also ein weiteres Systemelement des Reinforcement Learning, die ***State-Value-Funktion*** $V(s)$, welche mit Bezug auf eine Policy Π auch oft als $V_\Pi(s)$ bezeichnet wird: Denn ob eine Situation schlecht ist, hängt ja auch davon ab, was der Agent für ein Allgemeinverhalten Π an den Tag legt.

Eine Situation, die unter einer risikosuchenden, Grenzen austestenden Policy schlecht ist, wäre beispielsweise, wenn einem Agent auf einem Fahrrad das Vorderrad in der Kurve anfängt wegzurutschen und er in dieser Situation aufgrund seiner Draufgänger-Policy nicht bremst. Mit einer risikobewussten Policy sähe dieselbe Situation schon viel besser aus, würde also von einer guten State-Value-Funktion höher bewertet werden.

$V_\Pi(s)$ gibt einfach den Wert zurück, den die aktuelle Situation s unter der Policy Π für den Agenten gerade hat. Abstrakt nach den obigen Definitionen gesagt, entspricht der Wert der State-Value-Funktion dem Return R_t (dem erwarteten Wert) einer Situation s_t. E_Π bezeichnet hierbei die Menge der erwarteten Returns unter Π und der aktuellen Situation s_t.

$$V_\Pi(s) = E_\Pi\{R_t|s = s_t\}$$

-6	-5	-4	-3	-2
-7				-1
-6	-5	-4	-3	-2
-7	-6	-5		-3
-8	-7	-6		-4
-9	-8	-7		-5
-10	-9	-8	-7	-6

-6	-5	-4	-3	-2
-7				-1
-8	-9	-10		-2
-9	-10	-11		-3
-10	-11	-10		-4
-11	-10	-9		-5
-10	-9	-8	-7	-6

Abbildung C.3: Darstellung des jeweils optimalen Returns pro Feld in unserer Gridworld unter der Pure Negative Reward-Vergabe, oben mit offener Tür und unten mit geschlossener.

Definition C.9 (State-Value-Funktion)**.** Die State-Value-Funktion $V_\Pi(s)$ hat zur Aufgabe, den Wert von Situationen unter einer Policy zu ermitteln, also dem Agenten die Frage zu beantworten, ob eine Situation s gut oder schlecht ist oder wie gut bzw. schlecht sie ist. Hierfür gibt sie den Erwartungswert des Returns unter der Situation aus:

$$V_\Pi(s) = E_\Pi\{R_t | s = s_t\} \tag{C.13}$$

Die optimale State-Value-Funktion nennen wir $V_\Pi^*(s)$.

Nun hat unser Roboter im Gegensatz zu uns leider keine gottgleiche Sicht auf seine Umwelt. Er besitzt keine Tabelle mit optimalen Returns, wie wir sie eben aufgezeichnet haben, an der er sich orientieren könnte. Das Ziel von Reinforcement Learning ist es, dass der Roboter sich seine State-Value-Funktion anhand der Returns aus vielen Versuchen nach und nach selbst aufbaut und der optimalen State-Value-Funktion V^* annähert (wenn es eine gibt).

In diesem Zusammenhang seien noch zwei Begriffe eingeführt, welche eng mit dem Kreislauf zwischen State-Value-Funktion und Policy verbunden sind:

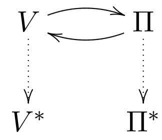


Abbildung C.4: Der Kreislauf des Reinforcement-Learnings, welcher idealerweise zu optimalem Π^* bzw. V^* führt.

C.2.2.1 Policy evaluation

Policy evaluation nennt man die Vorgehensweise, eine Policy einige Male durchzuprobieren, auf diese Weise viele Rewards zu erhalten und durch diese mit der Zeit eine State-Value-Funktion aufzukumulieren.

C.2.2.2 Policy improvement

Policy improvement bedeutet, eine Policy selbst zu verbessern, also aus ihr eine neue, bessere Policy zu erzeugen. Um die Policy zu verbessern, müssen wir das Ziel verfolgen, dass der Return zum Schluss einen größeren Wert hat als vorher – man also einen kürzeren Weg zum Restaurant gefunden hat und ihn auch erfolgreich gegangen ist.

Das Prinzip des Reinforcement Learnings ist nun, ein Wechselspiel zu realisieren. Man versucht zu bewerten, wie gut eine Policy in den einzelnen Situationen ist. Wir probieren eine Policy und erhalten eine veränderte State-Value-Funktion. Aus der Veränderung der State-Value-Funktion gewinnen wir Informationen über das System, aus der wir wieder unsere Policy verbessern. Diese beiden Werte ziehen sich nun gegenseitig hoch, was sich auch mathematisch beweisen lässt – so dass man zum Schluß eine optimale Policy Π^* und eine optimale State-Value-Funktion V^* erhält (Abb. C.4). Dieser Kreislauf hört sich einfach an, ist aber sehr langwierig.

Betrachten wir nun zuerst eine einfache, zufällige Policy, wie unser Roboter seine State-Value-Funktion ohne Vorwissen langsam ausfüllen und verbessern könnte.

C.2.3 Montecarlo-Methodik

Die einfachste Taktik, eine State-Value-Funktion aufzukumulieren, ist das reine Ausprobieren. Wir wählen also eine sich rein zufällig verhaltende Policy, welche die aufkumulierte State-Value-Funktion für ihre Zufallsentscheidungen nicht berücksichtigt. Es lässt sich beweisen, dass wir in unserer Gridworld irgendwann einmal durch Zufall den Ausgang finden werden.

Angelehnt an die auf Zufall basierenden Glücksspiele nennen wir diese Vorgehensweise ***Montecarlo-Methodik***.

Gehen wir weiterhin von einem *Pure Negative Reward* aus, so ist klar, dass wir für unser Startfeld in der State-Value-Funktion einen Bestwert von -6 erhalten können. Je nachdem, welchen zufälligen Weg die zufällige Policy aber einschlägt, können andere (kleinere) Werte als -6 für das Startfeld auftreten. Intuitiv möchten wir uns für einen Zustand (also ein Feld) jeweils nur den *besseren* Wert merken. Hier ist jedoch Vorsicht geboten: So würde das Lernverfahren nur bei *deterministischen Systemen funktionieren*. Unsere Tür, die pro Durchlauf entweder offen oder geschlossen sein kann, würde Oszillationen bei allen Feldern hervorrufen, deren kürzester Weg zum Ziel durch sie beeinflusst wird.

Wir verwenden bei der Montecarlo-Methodik also lieber die Lernregel[2]

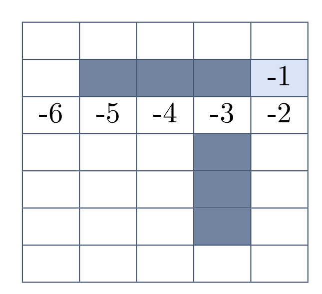
$$V(s_t)_{\text{neu}} = V(s_t)_{\text{alt}} + \alpha(R_t - V(s_t)_{\text{alt}}),$$

in der offensichtlich sowohl der alte Zustandswert als auch der erhaltene Return Einfluss auf die Aktualisierung der State-Value-Funktion haben (α ist die Lernrate). Der Agent erhält also eine Art Erinnerungsvermögen, neue Erkenntnisse ändern immer nur ein wenig am Situationswert. Ein beispielhafter Lernschritt findet sich in Abb. C.5 auf der folgenden Seite.

In dem Beispielbild wurde nur die Zustandswertberechnung für einen einzigen Zustand (unseren Startzustand) aufgetragen. Dass es möglich ist und auch oft gemacht wird, die Werte für die zwischendurch besuchten Zustände (im Fall der Gridworld unsere Wege zum Ziel) gleich mit zu trainieren, sollte offensichtlich sein. Das Ergebnis einer solchen Rechnung in Bezug auf unser Beispiel findet sich in Abb. C.6 auf Seite 251.

Die Montecarlo-Methodik mag suboptimal erscheinen und ist auch im Regelfall wesentlich langsamer als die nachfolgend vorgestellten Methoden des Reinforcement Learnings - allerdings ist sie die einzige, bei der man *mathematisch beweisen* kann, dass sie funktioniert und eignet sich daher sehr für theoretische Betrachtungen.

2 Sie wird u.a. unter Verwendung der *Bellman-Gleichung* hergeleitet, die Herleitung ist aber nicht Bestandteil des Kapitels.



				-1
-14	-13	-12		-2
		-11		-3
		-10		-4
		-9		-5
		-8	-7	-6

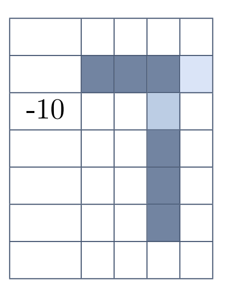


Abbildung C.5: Anwendung der Montecarlo-Lernregel mit einer Lernrate von $\alpha = 0.5$. Oben sind beispielhaft zwei Wege aufgetragen, die der Agent durch Zufall wählt (einer mit offener, einer mit geschlossener Tür). Unten das Ergebnis der Lernregel für den Wert des Startzustands unter Berücksichtigung beider Wege. Dadurch, dass unter zufälliger Policy mit der Zeit sehr viele verschiedene Wege gegangen werden, erhält man eine sehr aussagekräftige State-Value-Funktion.

				-1
-10	-9	-8	-3	-2
		-11		-3
		-10		-4
		-9		-5
		-8	-7	-6

Abbildung C.6: Erweiterung des Lernbeispiels aus Abb. C.5, in dem auch die Returns für Zwischenzustände zur Aufkumulierung der State-Value-Funktion herangezogen werden. Sehr schön zu beobachten ist der niedrige Wert auf dem Türfeld: Wenn dieser Zustand eingenommen werden kann, ist er ja sehr positiv, wenn die Tür zu ist, kann er gar nicht eingenommen werden.

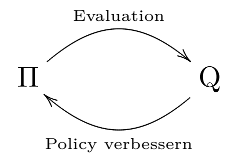


Abbildung C.7: Wir probieren Aktionen in der Umwelt aus und lernen so und verbessern die Policy.

Definition C.10 (Montecarlo-Lernen)**.** Es werden zufällig Aktionen ohne Rücksicht auf die State-Value-Funktion ausgeführt und langfristig eine aussagekräftige State-Value-Funktion mit untenstehender Lernregel aufkumuliert.

$$V(s_t)_{\text{neu}} = V(s_t)_{\text{alt}} + \alpha(R_t - V(s_t)_{\text{alt}}),$$

C.2.4 Temporal Difference Learning

Wir lernen im täglichen Leben weitestgehend durch Erfahrung und Ausprobieren. Das Allermeiste, was wir lernen, geschieht durch Erfahrung; blessurenfrei (oder eben auch nicht) gehen und Fahrrad fahren, auch geistige Fertigkeiten wie mathematisches Problemlösen profitieren sehr von Erfahrung und schlichtem Ausprobieren (*Trial and Error*). Wir initialisieren also unsere Policy mit irgendwelchen Werten - probieren aus, lernen und verbessern die Policy so *aus Erfahrung* (Abb. C.7). Im Unterschied zur Montecarlo-Methodik wollen wir dies nun auf eine gerichtetere Art und Weise tun.

Genau wie wir durch Erfahrung lernen, in verschiedenen Situationen auf bestimmte Weise zu reagieren, macht es die ***Temporal Difference*** Lernmethode (kurz: ***TD-Learning***), indem $V_\Pi(s)$ trainiert wird (der Agent lernt also einschätzen, welche Situationen viel wert sind und welche nicht). Wir bezeichnen wieder die aktuelle Situation mit s_t, die nachfolgende Situation mit s_{t+1} und so weiter. Die Lernformel für die State-Value-Funktion $V_\Pi(s_t)$ ergibt sich also zu

$$V(s_t)_{\text{neu}} = V(s_t) + \underbrace{\alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))}_{\text{Veränderung des alten Wertes}}$$

Wir sehen, dass die zur Lernrate α proportionale Veränderung des Wertes der aktuellen Situation s_t beeinflusst wird von

- dem empfangenen Reward r_{t+1},
- dem mit einem Faktor γ gewichteten bisherigen Return der Nachfolgesituation $V(s_{t+1})$,
- dem alten Wert der Situation $V(s_t)$.

Definition C.11 (Temporal Difference Learning)**.** Im Unterschied zur Montecarlo-Methodik schaut TD-Learning etwas in die Zukunft, indem die Nachfolgesituation s_{t+1} betrachtet wird. Die Lernregel ist definiert zu

$$V(s_t)_{\text{neu}} = V(s_t) + \underbrace{\alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))}_{\text{Veränderung des alten Wertes}}. \tag{C.14}$$

C.2.5 Die Action-Value-Funktion

Analog zur State-Value-Funktion $V_\Pi(s)$ ist ein weiterer Systembestandteil des Reinforcement Learning die ***Action-Value-Funktion*** $Q_\Pi(s, a)$, welche eine bestimmte Aktion a unter einer bestimmten Situation s und der Policy Π bewertet.

In der Gridworld: In der Gridworld sagt uns die Action-Value-Funktion, wie gut es ist, von einem bestimmten Feld in eine bestimmte Richtung zu gehen (Abb. C.8 auf der rechten Seite).

Definition C.12 (Action-Value-Funktion)**.** Analog zur State-Value-Funktion bewertet die Action-Value-Funktion $Q_\Pi(s_t, a)$ bestimmte Aktionen ausgehend von bestimmten Situationen unter einer Policy. Die optimale Action-Value-Funktion benennen wir mit $Q^*_\Pi(s_t, a)$.

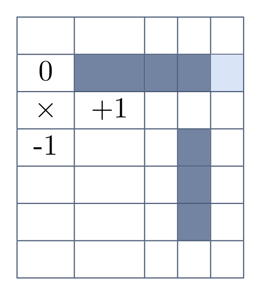


Abbildung C.8: Beispielhafte Werte einer Action-Value-Funktion für die Position ×. Nach rechts gehend bleibt man auf dem schnellsten Weg zum Ziel, nach oben ist immer noch ein recht schneller Weg, nach unten zu gehen ist kein guter Weg (alles im Falle einer offenen Tür).

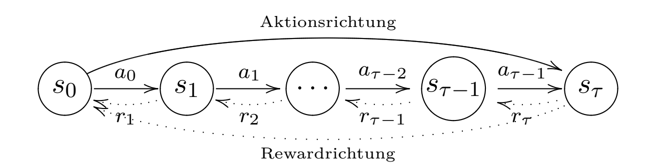


Abbildung C.9: Aktionen werden durchgeführt, bis eine gewünschte Zielsituation erreicht ist. Zu beachten ist die Durchnummerierung: Rewards werden von 1 an nummeriert, Aktionen und Situationen von 0 an (Dies hat sich einfach eingebürgert).

Wie in Abb. C.9 ersichtlich, führen wir Aktionen so lange durch, bis eine Zielsituation (hier s_τ genannt) erreicht ist (wenn es eine gibt, ansonsten werden einfach immer weiter Aktionen durchgeführt).

C.2.6 Q-Learning

Recht analog ergibt sich als Lernformel für die Action-Value-Funktion $Q_\Pi(s, a)$, deren Benutzung wir analog zum TD-Learning als ***Q-Learning*** bezeichnen:

$$
\begin{aligned}
Q(s_t, a)_{\text{neu}} = & Q(s_t, a) \\
& + \underbrace{\alpha(r_{t+1} + \gamma \underbrace{\max_a Q(s_{t+1}, a)}_{\text{Greedy-Strategie}} - Q(s_t, a))}_{\text{Veränderung des alten Wertes}}.
\end{aligned}
$$

Wir schlüsseln wieder die (zur Lernrate α proportionale) Veränderung des aktuellen Aktionswertes unter der aktuellen Situation auf. Sie wird beeinflusst von

- ▷ dem empfangenen Reward r_{t+1},
- ▷ dem mit γ gewichteten maximalen Action Value über die Nachfolgeaktionen (hier wird eine Greedy-Strategie angewendet, da man ruhig davon ausgehen kann, dass man die beste bekannte Aktion wählt, beim TD-Learning achten wir hingegen nicht darauf, immer in die beste bekannte nächste Situation zu kommen),
- ▷ dem alten Wert der Aktion unter unserer als s_t bekannten Situation $Q(s_t, a)$ (nicht vergessen, dass auch dieser durch α gewichtet ist).

In aller Regel lernt die Action-Value-Funktion wesentlich schneller als die State-Value-Funktion, wobei aber nicht zu vernachlässigen ist, dass Reinforcement Learning allgemein recht langsam ist: Das System muss ja selbst herausfinden, was gut ist. Schön ist aber am Q-Learning: Π kann beliebig initialisiert werden, durch Q-Learning erhalten wir *immer* Q^*.

Definition C.13 (Q-Learning)**.** Q-Learning trainiert die Action-Value-Funktion mit der Lernregel

$$\begin{aligned} Q(s_t, a)_{\text{neu}} =& Q(s_t, a) \\ &+ \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a)). \end{aligned} \tag{C.15}$$

und findet so auf jeden Fall Q^*.

C.3 Beispielanwendungen

C.3.1 TD-Gammon

TD-Gammon ist ein sehr erfolgreiches, auf TD-Learning basierendes Backgammonspiel von Gerald Tesauro. Situation ist hier die aktuelle Konfiguration des Spielbrettes. Jedem, der schon einmal Backgammon gespielt hat, ist klar, dass der Situationsraum gigantisch groß ist (ca. 10^{20} Situationen) – was zur Folge hat, dass man keine State-Value-Funktione explizit ausrechnen kann (insbesondere zur Zeit des TD-Gammons Ende der 80er Jahre). Die gewählte Rewardstrategie war *Pure Delayed Reward*, das System erhält den Reward also erst zum Schluss des Spiels, der dann gleichzeitig der Return ist. Man hat das System dann selbstständig üben lassen (zunächst

gegen ein Backgammonprogramm, danach gegen eine Instanz seiner selbst). Resultat war, das es in einer Computer-Backgammon-Liga das höchste Ranking erreicht hat und eindrucksvoll widerlegt war, dass ein Computerprogramm eine Tätigkeit nicht besser beherrschen kann als sein Programmierer.

C.3.2 Das Auto in der Grube

Betrachten wir ein Auto, das auf eindimensionaler Fahrbahn am Fuß einer tiefen Mulde steht, und die Steigung zu keiner der beiden Seiten auf Anhieb mit seiner Motorkraft überwinden kann, um aus der Mulde hinauszufahren. Ausführbare Aktionen sind hier trivialerweise die Möglichkeiten, vorwärts und rückwärts zu fahren. Die intuitive Lösung, an welche wir als Menschen sofort denken, ist zurückzusetzen, an der gegenüberliegenden Steigung Schwung zu holen und mehrmals auf diese Weise hin- und her zu oszillieren, um mit Schwung aus der Mulde herauszufahren.

Aktionen eines Reinforcement Learning Systems wären „Vollgas nach vorn“, „Vollgas zurück“ und „nichts tun“.

Hier wäre „Alles kostet“ eine gute Wahl für die Rewardvergabe, so dass das System schnell lernt, aus der Grube herauszukommen und merkt, dass unser Problem mit purer vorwärts gerichteter Motorkraft nicht zu lösen ist. Das System wird sich also langsam hochschaukeln.

Hier können wir die Policy nicht mehr als Tabelle ablegen, da der Zustandsraum nicht gut diskretisierbar ist. Hier muss wirklich eine Funktion als Policy geschaffen werden.

C.3.3 Der Pole Balancer

Der ***Pole Balancer*** wurde entwickelt von Barto, Sutton und Anderson.

Gegeben sei eine Situation, welche ein Fahrzeug beinhaltet, das sich entweder mit Vollgas nach rechts oder mit Vollgas nach links bewegen kann (Bang-Bang-Control). Es kann *nur* diese beiden Aktionen ausführen, Stehenbleiben ist nicht möglich. Auf diesem Fahrzeug steht senkrecht eine Stange, welche zu beiden Seiten umkippen kann. Die Stange ist so konstruiert, dass sie immer in Richtung einer Seite kippt, also niemals stillsteht (sagen wir einfach, sie sei am unteren Ende abgerundet).

Den Winkel, in dem die Stange im Moment relativ zur Senkrechten steht, bezeichnen wir mit α. Das Fahrzeug besitzt weiterhin immer eine definierte Position x auf unserer eindimensionalen Welt und immer eine Geschwindigkeit $\dot{x}$. Unsere eindimensionale Welt ist begrenzt, es gibt also Maximal- und Minimalwerte, welche x annehmen kann.

Ziel unseres Systems ist zu lernen, den Wagen dahingehend zu steuern, dass er das Kippen der Stange ausgleicht, die Stange also nicht umfällt. Dies erreichen wir am besten mit einer Avoidance Strategy: Solange die Stange nicht umgefallen ist, gibt es einen Reward von 0, fällt sie um, gibt es einen Reward von -1.

Interessanterweise ist das System schnell in der Lage, den Stab stehend zu halten, indem es schnell genug mit kleinen Bewegungen daran wackelt. Es hält sich hierbei zumeist in der Mitte des Raums auf, da es am weitesten weg von den Wänden ist, die es als negativ empfindet (stößt man gegen die Wand, fällt der Stab um).

C.3.3.1 Swinging up an inverted Pendulum

Schwieriger für das System ist die Startsituation, dass der Stab im Vorhinein herunterhängt, erst einmal durch Schwingungen über das Gefährt bewegt werden, und anschließend stabilisiert werden muss. Diese Aufgabe wird in der Literatur mit ***Swing up an inverted Pendulum*** bezeichnet.

C.4 Reinforcement Learning im Zusammenhang mit Neuronalen Netzen

Zu guter Letzt möchte der Leser vielleicht fragen, was das Kapitel über Reinforcement Learning in einem Skriptum zum Thema „Neuronale Netze“ zu suchen hat.

Die Antwort ist einfach motiviert. Wir haben bereits überwachte und unüberwachte Lernverfahren kennengelernt. Wir haben zwar nicht überall einen allwissenden Teacher, der uns überwachtes Lernen ermöglicht. Es ist aber auch nicht unbedingt so, dass wir *gar kein* Feedback erhalten. Oft gibt es ein Mittelding, eine Art Kritik oder Schulnote, bei Problemen dieser Art kann Reinforcement Learning zum Einsatz kommen.

Nicht alle Probleme sind hierbei so leicht handhabbar wie unsere Gridworld: Bei unserem Backgammon-Beispiel haben wir alleine ca. 10^{20} Situationen und einen großen Verzweigungsgrad im Situationsbaum, von anderen Spielen ganz zu schweigen. Hier werden die in der Gridworld benutzten Tabellen als State- und Action-Value-Funktionen schlicht nicht mehr realisierbar, wir müssen also Approximatoren für diese Funktionen finden.

Und welche lernfähigen Approximatoren für diese Bestandteile des Reinforcement Learnings fallen uns nun auf Anhieb ein? Richtig: Neuronale Netze.

Übungsaufgaben

Aufgabe 20. Ein Kontrollsystem für einen Roboter soll mittels Reinforcement Learning dazu gebracht werden, eine Strategie zu finden um ein Labyrinth möglichst schnell zu verlassen.

- Wie könnte eine geeignete State-Value-Funktion aussehen?
- Wie würden Sie einen geeigneten Reward erzeugen?

Gehen Sie davon aus, dass der Roboter Hindernisvermeidung beherrscht und zu jedem Zeitpunkt seine Position (x, y) und Orientierung ϕ kennt.

Aufgabe 21. Beschreiben Sie die Funktion der beiden Elemente *ASE* und *ACE* so wie sie von Barto, Sutton und Anderson für die Kontrolle des *Pole Balancer* vorgeschlagen wurden.

Literaturangabe: [BSA83].

Aufgabe 22. Nennen Sie mehrere „klassische“ Informatik-Aufgaben, die mit Reinforcement Learning besonders gut bearbeitet werden könnten und begründen Sie ihre Meinung.

Literaturverzeichnis

[And72] James A. Anderson. A simple neural network generating an interactive memory. *Mathematical Biosciences*, 14:197–220, 1972.

[APZ93] D. Anguita, G. Parodi, and R. Zunino. Speed improvement of the backpropagation on current-generation workstations. In *WCNN'93, Portland: World Congress on Neural Networks, July 11-15, 1993, Oregon Convention Center, Portland, Oregon*, volume 1. Lawrence Erlbaum, 1993.

[BSA83] A. Barto, R. Sutton, and C. Anderson. Neuron-like adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):834–846, September 1983.

[CG87] G. A. Carpenter and S. Grossberg. ART2: Self-organization of stable category recognition codes for analog input patterns. *Applied Optics*, 26:4919–4930, 1987.

[CG88] M.A. Cohen and S. Grossberg. Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *Computer Society Press Technology Series Neural Networks*, pages 70–81, 1988.

[CG90] G. A. Carpenter and S. Grossberg. ART 3: Hierarchical search using chemical transmitters in self-organising pattern recognition architectures. *Neural Networks*, 3(2):129–152, 1990.

[CH67] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[CR00] N.A. Campbell and JB Reece. *Biologie. Spektrum.* Akademischer Verlag, 2000.

[Cyb89] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, 1989.

[DHS01] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern classification.* Wiley New York, 2001.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, April 1990.

[Fah88] S. E. Fahlman. An empirical sudy of learning speed in back-propagation networks. Technical Report CMU-CS-88-162, CMU, 1988.

[FMI83] K. Fukushima, S. Miyake, and T. Ito. Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE Transactions on Systems, Man, and Cybernetics*, 13(5):826–834, September/October 1983.

[Fri94] B. Fritzke. Fast learning with incremental RBF networks. *Neural Processing Letters*, 1(1):2–5, 1994.

[GKE01a] N. Goerke, F. Kintzler, and R. Eckmiller. Self organized classification of chaotic domains from a nonlinearattractor. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, 2001.

[GKE01b] N. Goerke, F. Kintzler, and R. Eckmiller. Self organized partitioning of chaotic attractors for control. *Lecture notes in computer science*, pages 851–856, 2001.

[Gro76] S. Grossberg. Adaptive pattern classification and universal recoding, I: Parallel development and coding of neural feature detectors. *Biological Cybernetics*, 23:121–134, 1976.

[GS06] Nils Goerke and Alexandra Scherbart. Classification using multi-soms and multi-neural gas. In *IJCNN*, pages 3895–3902, 2006.

[Heb49] Donald O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, New York, 1949.

[Hop82] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Science, USA*, 79:2554–2558, 1982.

[Hop84] JJ Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, 81(10):3088–3092, 1984.

[HT85] JJ Hopfield and DW Tank. Neural computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152, 1985.

[Jor86] M. I. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Conference of the Cognitive Science Society*, pages 531–546. Erlbaum, 1986.

[Kau90] L. Kaufman. Finding groups in data: an introduction to cluster analysis. In *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley, New York, 1990.

[Koh72] T. Kohonen. Correlation matrix memories. *IEEEtC*, C-21:353–359, 1972.

[Koh82] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[Koh89] Teuvo Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, Berlin, third edition, 1989.

[Koh98] T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3):1–6, 1998.

[KSJ00] E.R. Kandel, J.H. Schwartz, and T.M. Jessell. *Principles of neural science*. Appleton & Lange, 2000.

[lCDS90] Y. le Cun, J. S. Denker, and S. A. Solla. Optimal brain damage. In D. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan Kaufmann, 1990.

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematics, Statistics and Probability, Vol. 1*, pages 281–296, 1967.

[MBS93] Thomas M. Martinetz, Stanislav G. Berkovich, and Klaus J. Schulten. 'Neural-gas' network for vector quantization and its application to time-series prediction. *IEEE Trans. on Neural Networks*, 4(4):558–569, 1993.

[MBW+10] K.D. Micheva, B. Busse, N.C. Weiler, N. O'Rourke, and S.J. Smith. Single-synapse analysis of a diverse synapse population: proteomic imaging methods and markers. *Neuron*, 68(4):639–653, 2010.

[MP43] W.S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 5(4):115–133, 1943.

[MP69] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, Mass, 1969.

[MR86] J. L. McClelland and D. E. Rumelhart. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, volume 2. MIT Press, Cambridge, 1986.

[Par87] David R. Parker. Optimal algorithms for adaptive networks: Second order back propagation, second order direct propagation, and second order hebbian learning. In Maureen Caudill and Charles Butler, editors, *IEEE First International Conference on Neural Networks (ICNN'87)*, volume II, pages II–593–II–600, San Diego, CA, June 1987. IEEE.

[PG89] T. Poggio and F. Girosi. *A theory of networks for approximation and learning.* MIT Press, Cambridge Mass., 1989.

[Pin87] F. J. Pineda. Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 59:2229–2232, 1987.

[PM47] W. Pitts and W.S. McCulloch. How we know universals the perception of auditory and visual forms. *Bulletin of Mathematical Biology*, 9(3):127–147, 1947.

[Pre94] L. Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. *Technical Report*, 21:94, 1994.

[RB93] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.

[RD05] G. Roth and U. Dicke. Evolution of the brain and intelligence. *Trends in Cognitive Sciences*, 9(5):250–257, 2005.

[RHW86a] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[RHW86b] David E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, and the PDP research group., editors, *Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundations.* MIT Press, 1986.

[Rie94] M. Riedmiller. Rprop - description and implementation details. Technical report, University of Karlsruhe, 1994.

[Ros58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

[Ros62] F. Rosenblatt. *Principles of Neurodynamics.* Spartan, New York, 1962.

[SB98] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[SG06] A. Scherbart and N. Goerke. Unsupervised system for discovering patterns in time-series, 2006.

[SGE05] Rolf Schatten, Nils Goerke, and Rolf Eckmiller. Regional and online learnable fields. In Sameer Singh, Maneesha Singh, Chidanand Apté, and Petra Perner, editors, *ICAPR (2)*, volume 3687 of *Lecture Notes in Computer Science*, pages 74–83. Springer, 2005.

[Ste61] K. Steinbuch. Die lernmatrix. *Kybernetik (Biological Cybernetics)*, 1:36–45, 1961.

[vdM73] C. von der Malsburg. Self-organizing of orientation sensitive cells in striate cortex. *Kybernetik*, 14:85–100, 1973.

[Was89] P. D. Wasserman. *Neural Computing Theory and Practice.* New York : Van Nostrand Reinhold, 1989.

[Wer74] P. J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, 1974.

[Wer88] P. J. Werbos. Backpropagation: Past and future. In *Proceedings ICNN-88, San Diego*, pages 343–353, 1988.

[WG94] A.S. Weigend and N.A. Gershenfeld. *Time series prediction.* Addison-Wesley, 1994.

[WH60] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *Proceedings WESCON*, pages 96–104, 1960.

[Wid89] R. Widner. Single-stage logic. AIEE Fall General Meeting, 1960. *Wasserman, P. Neural Computing, Theory and Practice, Van Nostrand Reinhold*, 1989.

[Zel94] Andreas Zell. *Simulation Neuronaler Netze.* Addison-Wesley, 1994. German.

Abbildungsverzeichnis

Index

*

A

B

C

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

Z